PYTHON

COMENZANDO A PROGRAMAR

LUIS MINGUILLÓN PASCUAL

Python

comenzando a programar

Luis Minguillón Pascual

PYTHON: COMENZANDO A PROGRAMAR
Autor: Luis Minguillón Pascual
© 2026 Luis Minguillón Pascual. Todos los derechos reservados.

Queda prohibida la reproducción total o parcial de esta obra, por cualquier medio o procedimiento, sin la autorización previa y por escrito del autor. Esto incluye la reproducción digital, mecánica, fotocopia, grabación, distribución o cualquier sistema de almacenamiento y recuperación de información.

El contenido de este libro se ofrece exclusivamente con fines educativos. El autor no se hace responsable del uso indebido del código, ejemplos o explicaciones aquí incluidos.

Primera edición: Zaragoza, España — 2026
ISBN: En trámite
Depósito legal: En trámite

Diseño, maquetación y edición: **Luis Minguillón Pascual**

Agradecimientos

Este libro, **Python: Comenzando a programar**, no habría sido posible sin el apoyo incondicional de las personas más importantes de mi vida.

Mi gratitud más profunda es para **Marisa**, mi mujer, cuya paciencia, cariño y fortaleza han sido el motor silencioso detrás de cada página escrita. Gracias por tu comprensión en las horas largas de trabajo, por tu ánimo constante y por recordarme siempre que los grandes proyectos se construyen con amor y dedicación. Este libro también es tuyo.

A mi **familia**, por estar siempre ahí, por su confianza en mí y por enseñarme desde pequeño el valor del esfuerzo y la curiosidad. Cada paso en este camino ha sido más fácil gracias a vuestro apoyo, vuestros consejos y vuestra presencia.

A todos vosotros, gracias por ser el pilar que sostiene mis ideas y el refugio al que siempre regreso.

Luis Minguillón Pascual
Zaragoza, 2026

Prefacio

Escribir este libro, **Python: Comenzando a programar**, ha sido un viaje que combina pasión, curiosidad y el deseo profundo de compartir conocimiento. La programación no es solo una herramienta técnica: es una forma de pensar, de resolver problemas y de comprender mejor el mundo digital que nos rodea. En una sociedad cada vez más conectada, aprender a programar se ha convertido en una habilidad esencial, tan importante como leer, escribir o razonar con lógica.

Python destaca por su sencillez, su potencia y su belleza. Es un lenguaje que invita a explorar, a experimentar y a crear. Su sintaxis clara y su enorme ecosistema lo convierten en el punto de partida ideal para quienes se acercan por primera vez al mundo del código. Por eso este libro nace con un objetivo muy concreto: **hacer que tu primer contacto con la programación sea accesible, comprensible y motivador**.

He escrito estas páginas pensando en quienes comienzan desde cero, en quienes sienten curiosidad pero no saben por dónde empezar, y también en quienes buscan una guía estructurada, directa y práctica. Aquí encontrarás explicaciones paso a paso, ejemplos claros y ejercicios diseñados para que avances con seguridad y confianza.

Creo firmemente que **el conocimiento debe compartirse**. No pertenece a quien lo posee, sino a quien lo necesita. Y también creo que **el conocimiento nos hace libres**: nos permite comprender, decidir, crear y transformar. Este libro es mi pequeña contribución a esa idea.

Si al terminarlo sientes que has dado tus primeros pasos en un camino nuevo, que has despertado una chispa de creatividad o que ahora entiendes un poco mejor cómo funciona el mundo digital, entonces este esfuerzo habrá merecido la pena.

Luis Minguillón Pascual
Zaragoza, 2026

INDICE:

- **Capítulo 1: Operadores en Python**
    - 1.1 Operadores aritméticos
    - 1.2 Operadores relacionales
    - 1.3 Operadores lógicos
    - 1.4 Operadores de asignación
    - 1.5 Operadores de identidad
    - 1.6 Operadores de pertenencia
    - 1.7 Precedencia y asociatividad
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 2: Tipos y Conversiones**
    - 2.2.1 De int a float y viceversa
    - 2.2.2 De número a cadena y viceversa
    - 2.7.2 ZeroDivisionError
    - 2.7.3 Problemas de precisión con float
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 3: Variables de cadena de texto en Python**
    - 3.1 Creación de cadenas
    - 3.2 Indexación y slicing
    - 3.3 Métodos fundamentales
    - 3.4 Búsqueda y reemplazo
    - 3.5 Formateo de cadenas
    - 3.6 Cadenas multilínea
    - 3.7 Codificación y Unicode
    - 3.8 Errores comunes (inmutabilidad, tipos, búsquedas)
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 4: Estructuras condicionales en Python**
    - 4.1 if
    - 4.2 if–else
    - 4.3 if–elif–else
    - 4.4 Condiciones múltiples
    - 4.5 Operadores lógicos
    - 4.6 Condiciones anidadas
    - 4.7 Buenas prácticas
    - 4.8 Errores frecuentes
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 5: Estructuras de repetición: bucles en Python**
    - 5.1 Bucle for
    - 5.2 Bucle while
    - 5.3 break, continue, pass
    - 5.4 Bucles anidados
    - 5.5 Iteración sobre colecciones
    - 5.6 List comprehensions

    - 5.7 Errores típicos
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 6: Listas en Python**
    - 6.1 Creación y características
    - 6.2 Indexación y slicing
    - 6.3 Métodos fundamentales
    - 6.4 Ordenación
    - 6.5 Copias superficiales y profundas
    - 6.6 Listas anidadas
    - 6.7 Errores comunes
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 7: Conjuntos (sets) en Python**
    - 7.1 Creación de sets
    - 7.2 Operaciones matemáticas
    - 7.3 Métodos principales
    - 7.4 Conjuntos inmutables (frozenset)
    - 7.5 Usos prácticos
    - 7.6 Errores típicos
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 8: Tuplas en Python**
    - 8.1 Creación y propiedades
    - 8.2 Indexación y slicing
    - 8.3 Desempaquetado
    - 8.4 Tuplas anidadas
    - 8.5 Usos prácticos
    - 8.6 Errores frecuentes
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 9: Diccionarios en Python**
    - 9.1 Creación de diccionarios
    - 9.2 Acceso a claves y valores
    - 9.3 Métodos fundamentales
    - 9.4 Diccionarios anidados
    - 9.5 Iteración sobre diccionarios
    - 9.6 Errores comunes
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 10: Funciones en Python**
    - 10.2 Parámetros y argumentos
    - 10.3 Valores por defecto
    - 10.4 *args y **kwargs
    - 10.5 Funciones lambda
    - 10.6 Ámbito de variables (local/global)
    - 10.7 Recursividad
    - 10.8 Buenas prácticas

    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 11: Programación Orientada a Objetos (POO) en Python**
    - 11.1 Conceptos fundamentales
    - 11.2 Clases y objetos
    - 11.3 Atributos y métodos
    - 11.4 Encapsulación
    - 11.5 Herencia
    - 11.6 Polimorfismo
    - 11.7 Métodos especiales (dunder methods)
    - 11.8 Buenas prácticas en POO
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 12: Tratamiento de Archivos y Ficheros en Python**
    - 12.3 Modos de apertura
    - 12.4 Gestión de rutas (os.path y pathlib)
    - 12.5 with open(): el gestor de contexto
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 13: Ficheros CSV en Python**
    - 13.0 Introducción a los ficheros CSV
    - 13.1 Lectura con csv.reader
    - 13.2 Escritura con csv.writer
    - 13.3 DictReader y DictWriter
    - 13.4 Delimitadores y dialectos
    - 13.5 Errores típicos con CSV
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 14: Ficheros JSON en Python**
    - 14.0 Introducción a JSON
    - 14.1 json.load y json.dump
    - 14.2 json.loads y json.dumps
    - 14.3 Correspondencia Diccionarios ↔ JSON
    - 14.4 Errores comunes y tipos no serializables
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 15: Tratamiento de Errores y Excepciones en Python**
    - 15.0 Introducción al tratamiento de errores
    - 15.4 Lanzamiento de excepciones (raise)
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 16: Formato de salida y color en consola**
    - 16.1 Códigos ANSI
    - 16.2 Colores básicos
    - 16.3 Estilos y fondos
    - 16.4 Librerías para color (colorama)
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 17: Formato de texto y números en Python**
    - 17.0 Introducción al formato

    - 17.1 El método format()
    - 17.2 f-strings
    - 17.3 Alineación
    - 17.4 Relleno
    - 17.5 Formato numérico (decimales, miles, porcentaje)
    - 17.6 Errores típicos
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 18: Módulos en Python**
    - 18.0 Introducción a los módulos
    - 18.1 La instrucción import
    - 18.2 from … import
    - 18.3 Creación de módulos propios
    - 18.4 Librerías estándar (math, random, os, etc.)
    - 18.5 Errores comunes al importar
    - **20 Ejercicios propuestos con soluciones**
- **Capítulo 19: Funciones avanzadas en Python**
    - 19.0 Introducción: funciones como ciudadanos de primera clase
    - 19.1 Funciones como objetos
    - 19.2 Decoradores
    - 19.3 Closures
    - 19.4 Funciones de orden superior (map, filter, sorted)
    - 19.5 Buenas prácticas avanzadas
    - **20 Ejercicios propuestos con soluciones**
- **ANEXO:**
    - A0 Introducción
    - A1 Poner título a la ventana de la consola
    - A2 Limpiar la consola
    - A3 Colocar texto en una posición concreta (ANSI)
    - A4 Randomize: semilla y aleatoriedad controlada
    - **20 Ejercicios propuestos con soluciones**

1. Operadores en Python

Los operadores son los “símbolos” que permiten que tu programa **haga cosas con los datos**: sumar, comparar, combinar condiciones, asignar valores, etc. Sin operadores, tu código solo almacenaría información, pero no podría **pensar ni decidir**.

En esta lección vamos a ver, con mucha calma y muchos ejemplos:

- **1.1 Operadores aritméticos**
- **1.2 Operadores relacionales**
- **1.3 Operadores lógicos**
- **1.4 Operadores de asignación**
- **1.5 Operadores de identidad**
- **1.6 Operadores de pertenencia**
- **1.7 Precedencia y asociatividad**
- **1.8 Evaluación paso a paso**
- **1.9 Errores comunes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

1.1 Operadores aritméticos

Los operadores aritméticos sirven para realizar **operaciones matemáticas** con números (enteros, flotantes) e incluso con otros tipos (por ejemplo, cadenas y listas en algunos casos).

Los principales operadores aritméticos en Python son:

- `+` → suma
- `-` → resta
- `*` → multiplicación
- `/` → división real (resultado flotante)
- `//` → división entera (suelo, “floor”)
- `%` → módulo (resto de la división)
- `**` → exponenciación (potencia) FreeCodecamp universopython.com

Ejemplos básicos

```
a = 10
b = 3

print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.3333333333333335
print(a // b)  # 3
print(a % b)   # 1
print(a ** b)  # 1000
```

Detalles importantes:

- `/` siempre devuelve un **float**, aunque la división sea exacta.

- // devuelve la parte entera, redondeando hacia abajo.
- **%** devuelve el **resto** de la división.
- ****** calcula potencias: `2 ** 10` es 1024.

Operadores aritméticos con otros tipos

En Python, algunos operadores aritméticos están **sobrecargados** para otros tipos:

```
# Suma de cadenas (concatenación)
s1 = "Hola "
s2 = "Python"
print(s1 + s2)  # "Hola Python"

# Multiplicación de cadenas por enteros (repetición)
print("ha" * 3)  # "hahaha"

# Listas
lista = [1, 2, 3]
print(lista * 2)  # [1, 2, 3, 1, 2, 3]
```

1.2 Operadores relacionales

Los operadores relacionales (o de comparación) sirven para **comparar valores** y devuelven siempre un **booleano**: `True` o `False`. FreeCodecamp universopython.com

Los principales son:

- == → igual que
- != → distinto de
- > → mayor que
- < → menor que
- >= → mayor o igual que
- <= → menor o igual que

Ejemplos básicos

```
x = 5
y = 8

print(x == y)   # False
print(x != y)   # True
print(x > y)    # False
print(x < y)    # True
print(x >= 5)   # True
print(y <= 8)   # True
```

Comparaciones con cadenas

Las cadenas se comparan **lexicográficamente** (como en un diccionario, usando el orden Unicode):

```
print("hola" == "hola")   # True
```

```
print("hola" == "Hola")   # False (mayúsculas/minúsculas importan)
print("a" < "b")          # True
print("perro" < "pez")    # True (se compara letra a letra)
```

1.3 Operadores lógicos

Los operadores lógicos permiten **combinar condiciones**. Trabajan con valores booleanos (`True` y `False`):

- **and** → verdadero si **ambas** condiciones son verdaderas
- **or** → verdadero si **al menos una** condición es verdadera
- **not** → niega el valor lógico (True → False, False → True) El Pythonista J2LOGO

Ejemplos básicos

```
edad = 20
tiene_carnet = True

print(edad >= 18 and tiene_carnet)  # True
print(edad < 18 or tiene_carnet)    # True
print(not tiene_carnet)             # False
```

Tablas de verdad (resumen mental)

- **and**
  - True and True → True
  - True and False → False
  - False and True → False
  - False and False → False
- **or**
  - True or True → True
  - True or False → True
  - False or True → True
  - False or False → False
- **not**
  - not True → False
  - not False → True

Cortocircuito (short-circuit)

Python **no evalúa** la segunda parte de la expresión si no es necesario:

```
def f():
    print("Evaluando f()")
    return True

# Aquí no se llama a f(), porque la primera parte ya es False
print(False and f())  # False
```

```
# Aquí sí se llama a f(), porque la primera parte es False y or necesita saber la segunda
print(False or f())   # Imprime "Evaluando f()" y luego True
```

1.4 Operadores de asignación

Los operadores de asignación sirven para **guardar valores en variables** y, en muchos casos, para **modificar** el valor actual de forma compacta. El Pythonista FreeCodecamp

El más básico es:

- `=` → asignación simple

Y los compuestos:

- `+=` → suma y asigna
- `-=` → resta y asigna
- `*****=**` → multiplica y asigna
- `/=` → divide y asigna
- `//=` → división entera y asigna
- **`%=`** → módulo y asigna
- **`**=`** → potencia y asigna

Ejemplos

```
x = 10      # asignación simple
x += 5      # x = x + 5  → x vale 15
x -= 3      # x = x - 3  → x vale 12
x *= 2      # x = x * 2  → x vale 24
x /= 4      # x = x / 4  → x vale 6.0 (float)
x //= 2     # x = x // 2 → x vale 3.0
x %= 2      # x = x % 2  → x vale 1.0
x **= 3     # x = x ** 3 → x vale 1.0
```

1.5 Operadores de identidad

Los operadores de identidad comparan **si dos variables apuntan al mismo objeto en memoria**, no si tienen el mismo valor. El Pythonista universopython.com

- **`is`** → es el mismo objeto
- **`is not`** → no es el mismo objeto

Ejemplo con números pequeños

```
a = 5
b = 5

print(a == b)  # True (mismo valor)
```

```
print(a is b)  # True (en muchos casos, mismo objeto internamente)
```

Ejemplo con listas

```
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
lista3 = lista1

print(lista1 == lista2)   # True (mismo contenido)
print(lista1 is lista2)   # False (objetos distintos)
print(lista1 is lista3)   # True (misma referencia)
```

Regla importante:
Usa `==` para comparar valores.
Usa **is** para comprobar identidad, especialmente con `None`:

```
x = None

if x is None:
    print("x no tiene valor")
```

1.6 Operadores de pertenencia

Los operadores de pertenencia comprueban si un elemento **está dentro** de una secuencia o colección (cadenas, listas, tuplas, diccionarios, conjuntos). El Pythonista J2LOGO

- **in** → el elemento pertenece
- **not in** → el elemento no pertenece

Ejemplos con listas

```
numeros = [1, 2, 3, 4, 5]

print(3 in numeros)      # True
print(10 in numeros)     # False
print(10 not in numeros) # True
```

Ejemplos con cadenas

```
texto = "Hola mundo"

print("Hola" in texto)     # True
print("hola" in texto)     # False (mayúsculas/minúsculas
importan)
print("mundo" in texto)    # True
print("Python" not in texto)  # True
```

Ejemplo con diccionarios

En diccionarios, `in` comprueba las **claves**:

```
persona = {"nombre": "Ana", "edad": 30}

print("nombre" in persona)   # True
print("Ana" in persona)      # False (no es clave, es valor)
print("edad" in persona)     # True
```

1.7 Precedencia y asociatividad

Cuando escribes una expresión con **muchos operadores**, Python necesita saber **en qué orden** evaluarlos. Eso se llama **precedencia de operadores**. universopython.com J2LOGO

Además, cuando varios operadores tienen la **misma precedencia**, entra en juego la **asociatividad** (normalmente de izquierda a derecha).

Precedencia simplificada (de mayor a menor)

1. ****** (potencia)
2. **+x, -x** (signo unario)
3. ********* , / , // , **%**
4. **+ , -** (suma y resta)
5. **<, <=, >, >=, ==, !=** (comparaciones)
6. **not**
7. **and**
8. **or**

Ejemplos de precedencia

```
resultado = 2 + 3 * 4
# Primero 3 * 4 = 12, luego 2 + 12 = 14
print(resultado)  # 14

resultado = (2 + 3) * 4
# Primero 2 + 3 = 5, luego 5 * 4 = 20
print(resultado)  # 20
```

Asociatividad

La mayoría de operadores binarios son **asociativos por la izquierda**:

```
print(10 - 3 - 2)
# Se evalúa como (10 - 3) - 2 = 7 - 2 = 5

print(2 ** 3 ** 2)
# La potencia es asociativa por la derecha:
# 3 ** 2 = 9, luego 2 ** 9 = 512
```

Regla práctica:
Cuando tengas dudas, **usa paréntesis** para dejar claro el orden.

1.8 Evaluación paso a paso

Veamos cómo Python evalúa expresiones **complejas** combinando varios tipos de operadores.

Ejemplo 1

```
expr = 3 + 4 * 2 > 10 and not (5 == 2 + 3)

# Paso 1: multiplicación
# 4 * 2 = 8
# expr = 3 + 8 > 10 and not (5 == 2 + 3)

# Paso 2: suma
# 3 + 8 = 11
# expr = 11 > 10 and not (5 == 2 + 3)

# Paso 3: comparación 11 > 10
# True
# expr = True and not (5 == 2 + 3)

# Paso 4: paréntesis (5 == 2 + 3)
# 2 + 3 = 5
# 5 == 5 → True
# expr = True and not True

# Paso 5: not True → False
# expr = True and False

# Paso 6: and
# True and False → False

print(expr)  # False
```

Ejemplo 2

```
a = 10
b = 5
c = 2

expr = a / b + c * (a - b) >= 5 and b != c

# Paso 1: paréntesis (a - b)
# (10 - 5) = 5
# expr = a / b + c * 5 >= 5 and b != c

# Paso 2: a / b
# 10 / 5 = 2.0
# expr = 2.0 + c * 5 >= 5 and b != c

# Paso 3: c * 5
# 2 * 5 = 10
# expr = 2.0 + 10 >= 5 and b != c

# Paso 4: suma
```

```
# 2.0 + 10 = 12.0
# expr = 12.0 >= 5 and b != c

# Paso 5: comparaciones
# 12.0 >= 5 → True
# b != c → 5 != 2 → True

# Paso 6: and
# True and True → True

print(expr)  # True
```

1.9 Errores comunes

1) Confundir = con ==

- = es asignación.
- == es comparación.

```
x = 5      # correcto
# if x = 5:  # ERROR de sintaxis
#     ...

if x == 5:  # correcto
    print("x vale 5")
```

2) Usar is en lugar de == para comparar valores

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True (mismo contenido)
print(a is b)  # False (objetos distintos)
```

3) Divisiones enteras inesperadas

```
print(5 / 2)   # 2.5
print(5 // 2)  # 2 (división entera)
```

4) Precedencia mal entendida

```
print(2 + 3 * 4)    # 14, no 20
print((2 + 3) * 4)  # 20
```

5) Comparar flotantes directamente

Por temas de precisión, a veces:

```
print(0.1 + 0.2 == 0.3)  # False en muchos casos
```

Mejor usar una **tolerancia**:

```
suma = 0.1 + 0.2
objetivo = 0.3
tolerancia = 1e-9

print(abs(suma - objetivo) < tolerancia)  # True
```

Ejercicios propuestos (20)

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1

Calculadora básica:
Pide dos números al usuario y muestra: suma, resta, multiplicación, división y módulo.

Ejercicio 2

División entera y resto:
Pide un número de segundos y muestra cuántos minutos y segundos son (usando `//` y `%`).

Ejercicio 3

Comparación de edades:
Pide dos edades y muestra si la primera es mayor, menor o igual que la segunda.

Ejercicio 4

Aprobado o suspenso:
Pide una nota (0–10) y muestra si está aprobada (≥ 5) o suspensa.

Ejercicio 5

Rango de edad:
Pide una edad y comprueba si está entre 18 y 65 años (ambos incluidos) usando operadores lógicos.

Ejercicio 6

Número par y múltiplo de 3:
Pide un número entero y muestra si es par y múltiplo de 3 a la vez.

Ejercicio 7

Descuento por compra:
Pide el importe de una compra y aplica un 10% de descuento si es mayor de 100, usando operadores relacionales y de asignación.

Ejercicio 8

Identidad con None:
Pide un texto al usuario. Si el usuario pulsa solo Enter, guarda `None` en la variable. Comprueba con `is` si la variable es `None`.

Ejercicio 9

Pertenencia en lista de usuarios:
Define una lista de nombres de usuarios permitidos. Pide un nombre y comprueba si está en la lista usando `in`.

Ejercicio 10

Pertenencia en cadena:
Pide una frase y una palabra, y comprueba si la palabra está dentro de la frase.

Ejercicio 11

Operadores combinados:
Pide un número entero y comprueba si está en el rango [1, 100] y además no es múltiplo de 5.

Ejercicio 12

Potencias y precedencia:
Calcula el valor de la expresión `2 + 3 * 2 ** 3` y luego el de `(2 + 3) * 2 ** 3`. Muestra ambos resultados.

Ejercicio 13

Actualización de saldo:
Tienes un saldo inicial de 1000. Pide un importe de ingreso y uno de gasto, y actualiza el saldo usando operadores de asignación (`+=` y `-=`).

Ejercicio 14

Comparación de cadenas:
Pide dos palabras y muestra si son iguales, si la primera es “menor” que la segunda o mayor (lexicográficamente).

Ejercicio 15

Uso de not:
Pide una edad y una respuesta de “tiene carnet” (sí/no). Usa `not` para mostrar un mensaje si la persona **no** puede conducir.

Ejercicio 16

Lista compartida o no:
Crea una lista `a` con algunos números. Asigna `b = a` y `c = a[:]`. Muestra el resultado de `a is b`, `a is c`, `a == c`.

Ejercicio 17

Contador con asignación compuesta:
Pide un número entero `n`. Empieza con `contador = 0` y ve sumando `n` tres veces usando `+=`. Muestra el resultado.

Ejercicio 18

Expresión lógica compleja:
Evalúa paso a paso la expresión:
`(5 > 3 and 2 == 2) or (10 < 5 and 3 != 3)`
Imprime el resultado final.

Ejercicio 19

Filtro de lista con pertenencia:
Define una lista de números. Pide un número y muestra si está en la lista. Si no está, añádelo a la lista y muestra la lista actualizada.

Ejercicio 20

Errores de precedencia:
Escribe una expresión que use `and`, `or` y comparaciones, y prueba a quitar y poner paréntesis para ver cómo cambia el resultado. Explica qué ocurre.

Soluciones de los ejercicios

Solución 1

```
a = float(input("Introduce el primer número: "))
b = float(input("Introduce el segundo número: "))

print("Suma:", a + b)
print("Resta:", a - b)
print("Multiplicación:", a * b)
print("División:", a / b)
print("Módulo:", a % b)
```

Solución 2

```
segundos = int(input("Introduce el número de segundos: "))

minutos = segundos // 60
resto = segundos % 60

print("Minutos:", minutos)
print("Segundos restantes:", resto)
```

Solución 3

```
edad1 = int(input("Edad 1: "))
edad2 = int(input("Edad 2: "))
```

```
if edad1 > edad2:
    print("La primera edad es mayor.")
elif edad1 < edad2:
    print("La primera edad es menor.")
else:
    print("Las dos edades son iguales.")
```

Solución 4

```
nota = float(input("Introduce la nota (0-10): "))

if nota >= 5:
    print("Aprobado")
else:
    print("Suspenso")
```

Solución 5

```
edad = int(input("Introduce tu edad: "))

if edad >= 18 and edad <= 65:
    print("Edad en rango laboral típico.")
else:
    print("Edad fuera del rango 18-65.")
```

Solución 6

```
n = int(input("Introduce un número entero: "))

es_par = (n % 2 == 0)
es_multiplo_3 = (n % 3 == 0)

if es_par and es_multiplo_3:
    print("El número es par y múltiplo de 3.")
else:
    print("No cumple ambas condiciones.")
```

Solución 7

```
importe = float(input("Importe de la compra: "))

if importe > 100:
    descuento = importe * 0.10
    importe -= descuento  # operador de asignación
    print("Se aplica un 10% de descuento.")
```

```
else:
    print("No se aplica descuento.")

print("Importe final:", importe)
```

Solución 8

```
texto = input("Introduce un texto (o pulsa Enter para dejarlo
vacío): ")

if texto == "":
    valor = None
else:
    valor = texto

if valor is None:
    print("No se introdujo texto.")
else:
    print("Texto introducido:", valor)
```

Solución 9

```
usuarios = ["ana", "luis", "maria", "juan"]

nombre = input("Nombre de usuario: ").lower()

if nombre in usuarios:
    print("Acceso permitido.")
else:
    print("Usuario no registrado.")
```

Solución 10

```
frase = input("Introduce una frase: ")
palabra = input("Introduce una palabra: ")

if palabra in frase:
    print("La palabra está en la frase.")
else:
    print("La palabra NO está en la frase.")
```

Solución 11

```
n = int(input("Introduce un número entero: "))
```

```
en_rango = (n >= 1 and n <= 100)
no_multiplo_5 = (n % 5 != 0)

if en_rango and no_multiplo_5:
    print("El número está en [1, 100] y NO es múltiplo de 5.")
else:
    print("No cumple las condiciones.")
```

Solución 12

```
expr1 = 2 + 3 * 2 ** 3
expr2 = (2 + 3) * 2 ** 3

print("2 + 3 * 2 ** 3 =", expr1)
print("(2 + 3) * 2 ** 3 =", expr2)
```

Solución 13

```
saldo = 1000.0

ingreso = float(input("Importe a ingresar: "))
gasto = float(input("Importe a gastar: "))

saldo += ingreso   # saldo = saldo + ingreso
saldo -= gasto     # saldo = saldo - gasto

print("Saldo final:", saldo)
```

Solución 14

```
pal1 = input("Primera palabra: ")
pal2 = input("Segunda palabra: ")

if pal1 == pal2:
    print("Las palabras son iguales.")
elif pal1 < pal2:
    print("La primera palabra es menor (lexicográficamente).")
else:
    print("La primera palabra es mayor (lexicográficamente).")
```

Solución 15

```
edad = int(input("Edad: "))
tiene_carnet = input("¿Tiene carnet? (s/n): ").lower() == "s"
```

```
puede_conducir = (edad >= 18 and tiene_carnet)

if not puede_conducir:
    print("No puede conducir.")
else:
    print("Puede conducir.")
```

Solución 16

```
a = [1, 2, 3]
b = a
c = a[:]

print("a is b:", a is b)   # True
print("a is c:", a is c)   # False
print("a == c:", a == c)   # True
```

Solución 17

```
n = int(input("Introduce un número entero: "))

contador = 0
contador += n
contador += n
contador += n

print("Resultado tras sumar n tres veces:", contador)
```

Solución 18

```
expr = (5 > 3 and 2 == 2) or (10 < 5 and 3 != 3)

# (5 > 3) → True
# (2 == 2) → True
# True and True → True
# (10 < 5) → False
# (3 != 3) → False
# False and False → False
# True or False → True

print(expr)  # True
```

Solución 19

```
numeros = [1, 3, 5, 7, 9]
```

```
n = int(input("Introduce un número: "))

if n in numeros:
    print("El número ya está en la lista.")
else:
    print("El número no está en la lista. Se añadirá.")
    numeros.append(n)

print("Lista actualizada:", numeros)
```

Solución 20

Ejemplo de código para experimentar:

```
a = True
b = False
c = True

expr1 = a or b and c
expr2 = (a or b) and c
expr3 = a or (b and c)

print("a or b and c:", expr1)
print("(a or b) and c:", expr2)
print("a or (b and c):", expr3)
```

- En `a or b and c`, primero se evalúa `b and c`, luego `a or resultado`.
- Cambiando los paréntesis, cambias el orden y, a veces, el resultado.

---

2. Variables numéricas en Python

Las **variables numéricas** son la base de casi todos los programas: cuentas, estadísticas, física, finanzas, juegos, simulaciones…
En Python trabajar con números es muy cómodo porque el lenguaje se encarga de muchos detalles internos (tamaño, memoria, etc.), pero es importante entender **qué tipos numéricos existen**, cómo se **convierten**, qué **operaciones** podemos hacer, cómo funcionan las **funciones numéricas integradas**, qué pasa con el **redondeo y la precisión**, cómo generar **números aleatorios** y cuáles son los **errores típicos** que suelen aparecer.

En esta lección veremos:

- **2.1 Tipos numéricos: int, float, complex**
- **2.2 Conversión entre tipos**
- **2.3 Operaciones matemáticas**
- **2.4 Funciones numéricas integradas**
- **2.5 Redondeo y precisión**
- **2.6 Números aleatorios**
- **2.7 Errores típicos**

Al final tendrás **20 ejercicios** con su **solución detallada**.

2.1 Tipos numéricos: int, float, complex

Python tiene tres tipos numéricos principales:

- **int** → números enteros (sin parte decimal)
- **float** → números en coma flotante (con parte decimal)
- **complex** → números complejos (parte real e imaginaria)

2.1.1 Enteros (int)

Un **entero** es un número sin decimales: …, -3, -2, -1, 0, 1, 2, 3, …

En Python:

```
a = 10
b = -5
c = 0


print(type(a))  # <class 'int'>
print(type(b))  # <class 'int'>
print(type(c))  # <class 'int'>
```

Características importantes de los int en Python:

- No tienen límite práctico de tamaño (más allá de la memoria disponible).
- Puedes trabajar con enteros muy grandes sin preocuparte por desbordamientos típicos de otros lenguajes.

```
n = 10**100  # 1 seguido de 100 ceros
print(n)
print(type(n))  # <class 'int'>
```

2.1.2 Flotantes (float)

Los **float** representan números reales aproximados, con parte decimal. Internamente se representan en **coma flotante binaria**, lo que provoca pequeñas imprecisiones.

```
x = 3.14
y = -0.001
z = 2.0

print(type(x))  # <class 'float'>
print(type(y))  # <class 'float'>
print(type(z))  # <class 'float'>
```

También puedes usar **notación científica**:

```
a = 1.5e3   # 1.5 * 10^3 = 1500.0
b = 2e-2    # 2 * 10^-2 = 0.02

print(a)  # 1500.0
print(b)  # 0.02
```

2.1.3 Complejos (complex)

Un número **complejo** tiene una parte real y una parte imaginaria. En Python se escribe con `j` para la parte imaginaria:

```
c1 = 2 + 3j
c2 = -1j
c3 = 5j

print(type(c1))  # <class 'complex'>
print(c1.real)   # 2.0
print(c1.imag)   # 3.0
```

Puedes operar con ellos:

```
a = 2 + 3j
b = 1 - 4j
```

```
print(a + b)  # (3-1j)
print(a * b)  # (14-5j)
```

Los complejos son muy útiles en **física**, **ingeniería**, **procesamiento de señales**, etc.

2.2 Conversión entre tipos

Muchas veces necesitarás **convertir** un número de un tipo a otro: de `int` a `float`, de `float` a `int`, de `str` a número, etc.

Las funciones principales son:

- **int(x)** → convierte a entero
- **float(x)** → convierte a flotante
- **complex(x)** o **complex(real, imag)** → convierte a complejo

2.2.1 De int a float y de float a int

```
a = 5      # int
b = float(a)
print(b, type(b))  # 5.0 <class 'float'>


c = 3.99   # float
d = int(c)
print(d, type(d))  # 3 <class 'int'>
```

Importante:
`int()` **trunca** la parte decimal (no redondea):

```
print(int(3.99))  # 3
print(int(-3.99)) # -3
```

2.2.2 De número a cadena y de cadena a número

Para mostrar números junto con texto, a menudo convertimos a cadena:

```
n = 42
texto = "El resultado es " + str(n)
print(texto)  # "El resultado es 42"
```

Para convertir una cadena a número:

```
s1 = "123"
s2 = "3.14"


n1 = int(s1)      # 123
```

```
n2 = float(s2)    # 3.14


print(n1, type(n1))
print(n2, type(n2))
```

Cuidado: si la cadena no tiene formato numérico válido, se produce un **ValueError**:

```
s = "abc"
# int(s)  # ValueError: invalid literal for int() with base 10: 'abc'
```

2.2.3 Conversión a complex

```
x = 5
y = 2.5


c1 = complex(x)       # (5+0j)
c2 = complex(x, y)    # (5+2.5j)


print(c1, type(c1))
print(c2, type(c2))
```

2.3 Operaciones matemáticas

Con números podemos hacer muchas operaciones. Empezamos por las **básicas** y luego veremos algunas más avanzadas.

2.3.1 Operadores aritméticos básicos

- `+` → suma
- `-` → resta
- `*****` → multiplicación
- `/` → división real (resultado float)
- `//` → división entera (suelo)
- **`%`** → módulo (resto)
- **`**`** → potencia

```
a = 10
b = 3


print(a + b)   # 13
print(a - b)   # 7
```

```
print(a * b)   # 30
print(a / b)   # 3.3333333333333335
print(a // b)  # 3
print(a % b)   # 1
print(a ** b)  # 1000
```

2.3.2 Operaciones mixtas entre int y float

Cuando mezclas `int` y `float`, el resultado suele ser `float`:

```
x = 5      # int
y = 2.0    # float

print(x + y)      # 7.0
print(type(x + y))  # <class 'float'>
```

2.3.3 Operaciones con complex

```
a = 2 + 3j
b = 1 - 4j

print(a + b)  # (3-1j)
print(a - b)  # (1+7j)
print(a * b)  # (14-5j)
print(a / b)  # (-0.5882352941176471+0.6470588235294118j)
```

2.4 Funciones numéricas integradas

Python incluye muchas **funciones integradas** (built-in) que trabajan con números. Algunas de las más usadas:

- **abs(x)** → valor absoluto
- **round(x, ndigits)** → redondeo
- **pow(x, y)** → potencia (equivalente a `x ** y`)
- **min(a, b, …)** → mínimo
- **max(a, b, …)** → máximo
- **sum(iterable)** → suma de elementos de un iterable

2.4.1 abs()

```
print(abs(5))     # 5
print(abs(-5))    # 5
print(abs(-3.14)) # 3.14
```

2.4.2 round()

```
x = 3.14159


print(round(x))      # 3
print(round(x, 2))   # 3.14
print(round(x, 3))   # 3.142
```

Nota: `round()` usa una regla de redondeo “al par” en el dígito final en algunos casos (por ejemplo, 2.5 puede redondear a 2 o 3 según el contexto). Lo veremos más en la sección de precisión.

2.4.3 pow()

```
print(pow(2, 3))   # 8
print(2 ** 3)      # 8
```

`pow()` también admite un tercer argumento para módulo: `pow(base, exp, mod)`.

```
print(pow(2, 10, 1000))  # (2**10) % 1000 = 24
```

2.4.4 min(), max(), sum()

```
numeros = [3, 7, 2, 9, 4]


print(min(numeros))  # 2
print(max(numeros))  # 9
print(sum(numeros))  # 25
```

2.5 Redondeo y precisión

Aquí está una de las partes **más delicadas** de los números en Python: la **precisión de los float**.

2.5.1 Problema típico de precisión

```
x = 0.1 + 0.2
print(x)           # 0.30000000000000004
print(x == 0.3)    # False
```

Esto no es un “fallo” de Python, sino una consecuencia de la representación en **coma flotante binaria**. Muchos números decimales no se pueden representar exactamente en binario.

2.5.2 Comparar flotantes con tolerancia

En lugar de comparar directamente `x == 0.3`, se suele usar una **tolerancia**:

```
x = 0.1 + 0.2
objetivo = 0.3
```

```
tolerancia = 1e-9

if abs(x - objetivo) < tolerancia:
    print("Son prácticamente iguales")
else:
    print("Son diferentes")
```

2.5.3 Redondeo con round()

```
x = 1.23456

print(round(x, 0))  # 1.0
print(round(x, 1))  # 1.2
print(round(x, 2))  # 1.23
print(round(x, 3))  # 1.235
```

2.5.4 Redondeo bancario (al par)

En algunos casos, `round()` aplica una regla llamada **"round half to even"** (redondeo al par):

```
print(round(2.5))  # 2
print(round(3.5))  # 4
```

Esto puede sorprender, pero es útil para evitar sesgos en grandes cantidades de datos.

2.6 Números aleatorios

Para trabajar con **números aleatorios** usamos el módulo **random** de la biblioteca estándar.

```
import random
```

2.6.1 random.random()

Devuelve un float en el rango [0.0, 1.0):

```
import random

x = random.random()
print(x)  # algo como 0.374829374
```

2.6.2 random.randint(a, b)

Devuelve un entero entre **a** y **b**, ambos incluidos:

```
import random
```

```
n = random.randint(1, 6)  # simular un dado
print(n)
```

2.6.3 random.uniform(a, b)

Devuelve un float entre **a** y **b**:

```
import random

x = random.uniform(1.0, 2.0)
print(x)  # por ejemplo 1.3748
```

2.6.4 random.choice() y random.shuffle()

Aunque no son estrictamente “numéricos”, se usan mucho con listas:

```
import random

lista = [1, 2, 3, 4, 5]

print(random.choice(lista))  # elige un elemento al azar

random.shuffle(lista)        # baraja la lista
print(lista)
```

2.7 Errores típicos

Trabajando con números es muy fácil cometer ciertos errores. Veamos los más frecuentes.

2.7.1 ValueError al convertir cadenas

```
s = "12a"
# n = int(s)  # ValueError: invalid literal for int() with base 10: '12a'
```

Solución: validar la entrada o usar `try/except`:

```
s = input("Introduce un número entero: ")

try:
    n = int(s)
    print("Has introducido:", n)
except ValueError:
```

```
    print("Eso no es un entero válido.")
```

2.7.2 ZeroDivisionError

Dividir entre cero produce un error:

```
a = 10
b = 0
# print(a / b)  # ZeroDivisionError: division by zero
```

Solución: comprobar antes:

```
if b != 0:
    print(a / b)
else:
    print("No se puede dividir entre cero.")
```

2.7.3 Problemas de precisión con float

Ya lo vimos: `0.1 + 0.2` no es exactamente `0.3`.
Solución: usar tolerancias o, si necesitas precisión decimal exacta (por ejemplo, en finanzas), usar el módulo `decimal` (tema más avanzado).

2.7.4 Usar int() esperando redondeo

```
print(int(3.9))  # 3, no 4
```

Solución: si quieres redondear, usa `round()`:

```
print(round(3.9))  # 4
```

2.7.5 Mezclar tipos sin querer

```
n = 5
texto = "El número es " + n  # TypeError
```

Solución: convertir a cadena:

```
n = 5
texto = "El número es " + str(n)
print(texto)
```

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Pide al usuario dos números enteros y muestra su suma, resta, producto y cociente (división real).

Ejercicio 2
Pide un número real (float) y muestra su valor absoluto, su cuadrado y su raíz cuadrada (usa `** 0.5`).

Ejercicio 3
Pide al usuario un número de metros (float) y conviértelo a centímetros y milímetros (usa operaciones aritméticas).

Ejercicio 4
Pide un número entero y muestra si es par o impar usando el operador `%`.

Ejercicio 5
Pide un número real y muestra su valor redondeado a 2 decimales usando `round()`.

Ejercicio 6
Pide al usuario el radio de un círculo (float) y calcula su área y su longitud de circunferencia (usa `pi = 3.14159`).

Ejercicio 7
Pide al usuario una cantidad en euros (float) y un tipo de cambio a dólares (float). Muestra el equivalente en dólares con 2 decimales.

Ejercicio 8
Pide al usuario un número entero y conviértelo a float. Luego pide un número en formato texto (por ejemplo `"3.14"`) y conviértelo a float. Muestra ambos resultados y sus tipos.

Ejercicio 9
Pide al usuario una cadena y trata de convertirla a entero con `int()`. Si falla, muestra un mensaje de error usando `try/except`.

Ejercicio 10
Genera un número entero aleatorio entre 1 y 100 y pídele al usuario que lo adivine. Indica si el número introducido es mayor, menor o igual que el número secreto.

Ejercicio 11
Genera tres números aleatorios entre 0 y 1 (float) y muestra su suma. ¿Puede ser mayor que 3? ¿Por qué?

Ejercicio 12
Pide al usuario la parte real y la parte imaginaria de un número complejo. Crea el número complejo y muestra su módulo (usa `abs()`).

Ejercicio 13
Pide al usuario un número real y muestra:

- Su parte entera (usando `int()`)
- Su parte decimal (restando la parte entera al número original)

Ejercicio 14
Pide al usuario dos números reales y muestra cuál es el mayor usando `max()` y cuál es el menor usando `min()`.

Ejercicio 15
Pide al usuario una lista de 5 números reales (puedes pedirlos uno a uno) y muestra su suma y su media (promedio).

Ejercicio 16
Pide al usuario un número real y muestra su valor redondeado a 0, 1, 2 y 3 decimales. Comenta las diferencias.

Ejercicio 17
Simula el lanzamiento de dos dados (números aleatorios entre 1 y 6) y muestra la suma de ambos. Repite el experimento 5 veces.

Ejercicio 18
Pide al usuario un número real y comprueba si está “cerca” de 10, es decir, si la diferencia absoluta con 10 es menor que 0.5.

Ejercicio 19
Pide al usuario un número entero `n` y calcula la suma de los números del 1 al `n` usando la fórmula `n * (n + 1) / 2`. Muestra el resultado como entero.

Ejercicio 20
Pide al usuario un número real y muestra si su parte decimal es mayor o menor que 0.5. Usa operaciones aritméticas y conversión a entero.

Soluciones de los 20 ejercicios

Solución 1

```
a = int(input("Introduce el primer entero: "))
b = int(input("Introduce el segundo entero: "))

suma = a + b
resta = a - b
producto = a * b
cociente = a / b  # suponemos que b != 0

print("Suma:", suma)
print("Resta:", resta)
print("Producto:", producto)
print("Cociente:", cociente)
```

Solución 2

```
x = float(input("Introduce un número real: "))

valor_absoluto = abs(x)
cuadrado = x ** 2
raiz = x ** 0.5 if x >= 0 else None

print("Valor absoluto:", valor_absoluto)
print("Cuadrado:", cuadrado)

if raiz is not None:
    print("Raíz cuadrada:", raiz)
else:
    print("No se puede calcular la raíz cuadrada real de un número negativo.")
```

Solución 3

```
metros = float(input("Introduce una cantidad en metros: "))

centimetros = metros * 100
milimetros = metros * 1000

print("Centímetros:", centimetros)
print("Milímetros:", milimetros)
```

Solución 4

```
n = int(input("Introduce un número entero: "))

if n % 2 == 0:
    print("El número es par.")
else:
    print("El número es impar.")
```

Solución 5

```
x = float(input("Introduce un número real: "))

redondeado = round(x, 2)

print("Número redondeado a 2 decimales:", redondeado)
```

Solución 6

```
pi = 3.14159

radio = float(input("Introduce el radio del círculo: "))

area = pi * radio ** 2
longitud = 2 * pi * radio

print("Área:", area)
print("Longitud de la circunferencia:", longitud)
```

Solución 7

```
euros = float(input("Cantidad en euros: "))
tipo_cambio = float(input("Tipo de cambio (1 euro = ? dólares): 
"))

dolares = euros * tipo_cambio

print("Equivalente en dólares: {:.2f}".format(dolares))
```

Solución 8

```
entero = int(input("Introduce un número entero: "))
entero_a_float = float(entero)
```

```
texto_num = input("Introduce un número en formato texto (por ejemplo 3.14): ")
float_desde_texto = float(texto_num)


print("Entero convertido a float:", entero_a_float, type(entero_a_float))
print("Texto convertido a float:", float_desde_texto, type(float_desde_texto))
```

Solución 9

```
s = input("Introduce una cadena que represente un entero: ")


try:
    n = int(s)
    print("Has introducido el entero:", n)
except ValueError:
    print("Error: la cadena no es un entero válido.")
```

Solución 10

```
import random

secreto = random.randint(1, 100)

intento = int(input("Adivina el número (entre 1 y 100): "))

if intento < secreto:
    print("Demasiado pequeño.")
elif intento > secreto:
    print("Demasiado grande.")
else:
    print("¡Has acertado!")
```

(Podrías mejorarlo con un bucle para permitir varios intentos.)

Solución 11

```
import random

a = random.random()
b = random.random()
c = random.random()

suma = a + b + c

print("a =", a)
print("b =", b)
print("c =", c)
print("Suma =", suma)
```

La suma **no puede ser mayor que 3**, porque cada número está en [0, 1). El máximo teórico sería 2.9999…

Solución 12

```
real = float(input("Parte real: "))
imag = float(input("Parte imaginaria: "))

z = complex(real, imag)
modulo = abs(z)

print("Número complejo:", z)
print("Módulo:", modulo)
```

Solución 13

```
x = float(input("Introduce un número real: "))

parte_entera = int(x)
parte_decimal = x - parte_entera
```

```
print("Parte entera:", parte_entera)
print("Parte decimal:", parte_decimal)
```

Solución 14

```
a = float(input("Introduce el primer número real: "))
b = float(input("Introduce el segundo número real: "))

mayor = max(a, b)
menor = min(a, b)

print("Mayor:", mayor)
print("Menor:", menor)
```

Solución 15

```
numeros = []

for i in range(5):
    x = float(input(f"Introduce el número {i+1}: "))
    numeros.append(x)

suma = sum(numeros)
media = suma / len(numeros)

print("Suma:", suma)
print("Media:", media)
```

Solución 16

```
x = float(input("Introduce un número real: "))

r0 = round(x, 0)
r1 = round(x, 1)
r2 = round(x, 2)
```

```
r3 = round(x, 3)

print("Redondeado a 0 decimales:", r0)
print("Redondeado a 1 decimal:", r1)
print("Redondeado a 2 decimales:", r2)
print("Redondeado a 3 decimales:", r3)
```

Puedes observar cómo cambia el valor según el número de decimales.

Solución 17

```
import random

for i in range(5):
    dado1 = random.randint(1, 6)
    dado2 = random.randint(1, 6)
    suma = dado1 + dado2
    print(f"Lanzamiento {i+1}: dado1 = {dado1}, dado2 = {dado2}, suma = {suma}")
```

Solución 18

```
x = float(input("Introduce un número real: "))

diferencia = abs(x - 10)

if diferencia < 0.5:
    print("El número está cerca de 10.")
else:
    print("El número NO está cerca de 10.")
```

Solución 19

```
n = int(input("Introduce un número entero n: "))

suma = n * (n + 1) / 2
```

```
suma_entera = int(suma)

print("La suma de 1 a n es:", suma_entera)
```

Solución 20

```
x = float(input("Introduce un número real: "))

parte_entera = int(x)
parte_decimal = x - parte_entera

if parte_decimal > 0.5:
    print("La parte decimal es mayor que 0.5.")
elif parte_decimal < 0.5:
    print("La parte decimal es menor que 0.5.")
else:
    print("La parte decimal es exactamente 0.5.")
```

---

3. Variables de cadena de texto en Python

Las **cadenas de texto** (strings) son uno de los tipos más importantes en Python. Todo lo que tenga que ver con nombres, mensajes, rutas de archivos, datos leídos de un fichero, respuestas de un usuario, etc., se representa normalmente como texto.

En esta lección vamos a ver, con mucha calma y muchos ejemplos:

- **3.1 Creación de cadenas**
- **3.2 Indexación y slicing**
- **3.3 Métodos fundamentales**
- **3.4 Búsqueda y reemplazo**
- **3.5 Formateo de cadenas**
- **3.6 Cadenas multilínea**
- **3.7 Codificación y Unicode**
- **3.8 Errores comunes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

3.1 Creación de cadenas

Una **cadena** es una secuencia ordenada de caracteres. En Python se representa entre **comillas simples** o **comillas dobles**:

```
s1 = "Hola"
s2 = 'Mundo'


print(s1)
print(s2)
```

Ambas formas son equivalentes. Lo importante es que **empieces y termines con el mismo tipo de comillas**.

También puedes crear cadenas vacías:

```
vacia1 = ""
vacia2 = ''


print(len(vacia1))  # 0
print(len(vacia2))  # 0
```

Caracteres especiales (como saltos de línea, tabulaciones, etc.) se representan con **secuencias de escape**, por ejemplo:

- `\n` → salto de línea
- `\t` → tabulación
- `\\` → barra invertida

- `\"` → comilla doble
- `\'` → comilla simple

```
s = "Línea 1\nLínea 2"
print(s)


t = "Columna1\tColumna2"
print(t)
```

También puedes usar **cadenas crudas** (raw strings) anteponiendo `r` para que no se interpreten las secuencias de escape:

```
ruta = r"C:\nueva\carpeta\archivo.txt"
print(ruta)  # Se muestra tal cual, sin interpretar \n como salto de línea
```

3.2 Indexación y slicing

Una cadena es una **secuencia** de caracteres, lo que significa que puedes acceder a cada carácter por su **posición** (índice).

Los índices empiezan en **0**:

```
texto = "Python"


#   P   y   t   h   o   n
#   0   1   2   3   4   5


print(texto[0])  # 'P'
print(texto[1])  # 'y'
print(texto[5])  # 'n'
```

También puedes usar índices **negativos**, que cuentan desde el final:

```
print(texto[-1])  # 'n'
print(texto[-2])  # 'o'
print(texto[-6])  # 'P'
```

Slicing (rebanado)

El **slicing** permite obtener **subcadenas** usando la sintaxis:

```
subcadena = texto[inicio:fin:paso]
```

- `inicio` → índice donde empieza (incluido)

- `fin` → índice donde termina (excluido)
- `paso` → salto entre índices (opcional)

Ejemplos:

```
texto = "Python"


print(texto[0:2])   # 'Py'  (índices 0 y 1)
print(texto[2:6])   # 'thon'
print(texto[:3])    # 'Pyt'  (desde el inicio hasta el índice 2)
print(texto[3:])    # 'hon'  (desde el índice 3 hasta el final)
print(texto[:])     # 'Python' (copia completa)
```

Con paso:

```
print(texto[0:6:2])  # 'Pto'  (índices 0, 2, 4)
print(texto[::2])    # 'Pto'
print(texto[::-1])   # 'nohtyP' (cadena invertida)
```

Importante: las cadenas en Python son **inmutables**, es decir, no puedes cambiar un carácter directamente:

```
texto = "Hola"
# texto[0] = "h"  # ERROR: TypeError
```

Para “modificar” una cadena, debes crear una nueva:

```
texto = "Hola"
nuevo = "h" + texto[1:]
print(nuevo)  # 'hola'
```

3.3 Métodos fundamentales

Las cadenas tienen muchos **métodos** que permiten transformarlas, analizarlas o consultarlas. Algunos de los más usados:

- **len(cadena)** → longitud
- **lower()** → todo en minúsculas
- **upper()** → todo en mayúsculas
- **capitalize()** → primera letra en mayúscula
- **title()** → primera letra de cada palabra en mayúscula
- **strip()** → elimina espacios en blanco al principio y al final
- **lstrip()**, **rstrip()** → eliminan espacios a la izquierda o derecha
- **replace(viejo, nuevo)** → reemplaza subcadenas
- **split(sep)** → divide en lista de subcadenas
- **join(iterable)** → une elementos con un separador

3.3.1 len()

```
texto = "Hola mundo"
print(len(texto))  # 10 (incluye el espacio)
```

3.3.2 lower(), upper(), capitalize(), title()

```
s = "hOlA mUnDo"

print(s.lower())      # 'hola mundo'
print(s.upper())      # 'HOLA MUNDO'
print(s.capitalize()) # 'Hola mundo'
print(s.title())      # 'Hola Mundo'
```

3.3.3 strip(), lstrip(), rstrip()

```
s = "   hola   "

print("[" + s + "]")
print("[" + s.strip() + "]")   # elimina espacios a ambos lados
print("[" + s.lstrip() + "]")  # elimina espacios a la izquierda
print("[" + s.rstrip() + "]")  # elimina espacios a la derecha
```

3.3.4 replace()

```
texto = "Me gusta Python. Python es genial."

nuevo = texto.replace("Python", "Java")
print(nuevo)  # 'Me gusta Java. Java es genial.'
```

Puedes limitar el número de reemplazos:

```
texto = "uno dos dos dos"
print(texto.replace("dos", "X", 2))  # 'uno X X dos'
```

3.3.5 split() y join()

```
frase = "uno dos tres cuatro"

palabras = frase.split()  # por defecto separa por espacios
print(palabras)  # ['uno', 'dos', 'tres', 'cuatro']

csv = "manzana,pera,plátano"
```

```
frutas = csv.split(",")
print(frutas)  # ['manzana', 'pera', 'plátano']
```

`join()` hace lo contrario: une una lista de cadenas con un separador:

```
palabras = ["Hola", "mundo", "Python"]
frase = " ".join(palabras)
print(frase)  # 'Hola mundo Python'

csv = ",".join(frutas)
print(csv)  # 'manzana,pera,plátano'
```

3.4 Búsqueda y reemplazo

Para **buscar** subcadenas dentro de una cadena, tenemos varios métodos:

- **find(sub)** → devuelve el índice de la primera aparición o -1 si no se encuentra
- **rfind(sub)** → igual que find, pero desde el final
- **index(sub)** → como find, pero lanza error si no se encuentra
- **count(sub)** → cuenta cuántas veces aparece la subcadena
- **replace(viejo, nuevo)** → ya visto, para reemplazar

3.4.1 find(), rfind(), index()

```
texto = "abracadabra"

print(texto.find("bra"))   # 1
print(texto.find("z"))     # -1

print(texto.rfind("bra"))  # 8

print(texto.index("bra"))  # 1
# print(texto.index("z"))  # ValueError
```

3.4.2 count()

```
texto = "banana"

print(texto.count("a"))   # 3
print(texto.count("na"))  # 2
```

3.4.3 Ejemplo combinado de búsqueda y reemplazo

```
texto = "El perro persigue al gato. El perro ladra."

veces = texto.count("perro")
print("La palabra 'perro' aparece", veces, "veces.")

nuevo = texto.replace("perro", "gato")
print(nuevo)
```

3.5 Formateo de cadenas

El **formateo** de cadenas consiste en **insertar valores** (números, otras cadenas, etc.) dentro de una plantilla de texto.

En Python hay varias formas:

- Operador **%** (estilo antiguo)
- Método **format()**
- **f-strings** (cadenas literales formateadas, desde Python 3.6)

3.5.1 Operador %

```
nombre = "Luis"
edad = 30

mensaje = "Me llamo %s y tengo %d años" % (nombre, edad)
print(mensaje)
```

- `%s` → cadena
- `%d` → entero
- `%f` → flotante

```
pi = 3.14159265
print("Pi vale aproximadamente %.2f" % pi)  # 2 decimales
```

3.5.2 format()

```
nombre = "Ana"
edad = 25

mensaje = "Me llamo {} y tengo {} años".format(nombre, edad)
print(mensaje)
```

```
mensaje = "Me llamo {0} y tengo {1} años. {0} es
programadora.".format(nombre, edad)
print(mensaje)
```

Con formato numérico:

```
pi = 3.14159265
print("Pi vale aproximadamente {:.3f}".format(pi))  # 3 decimales
```

3.5.3 f-strings

Son la forma más moderna y cómoda:

```
nombre = "Carlos"
edad = 40
pi = 3.14159265

mensaje = f"Me llamo {nombre} y tengo {edad} años."
print(mensaje)

print(f"Pi vale aproximadamente {pi:.4f}")
```

Dentro de las llaves `{}` puedes poner **expresiones**:

```
a = 5
b = 3
print(f"{a} + {b} = {a + b}")
```

3.6 Cadenas multilínea

Para escribir texto en **varias líneas** puedes usar **comillas triples**:

```
texto = """Esta es una cadena
que ocupa varias
líneas de código."""

print(texto)
```

También puedes usar comillas simples triples:

```
texto = '''Línea 1
Línea 2
```

```
Línea 3'''
print(texto)
```

Las cadenas multilínea son muy útiles para:

- Mensajes largos
- Documentación
- Plantillas de texto

3.7 Codificación y Unicode

Python 3 usa **Unicode** para representar cadenas, lo que permite trabajar con caracteres de casi todos los idiomas: tildes, eñes, símbolos, emojis, etc.

```
s = "Hola, ¿qué tal? 😊"
print(s)
print(len(s))  # cuenta caracteres, no bytes
```

Internamente, cuando se guarda en un archivo o se envía por la red, la cadena se **codifica** en una secuencia de bytes usando una **codificación** (por ejemplo, UTF-8).

3.7.1 encode() y decode()

- **encode(codificación)** → convierte la cadena en bytes
- **decode(codificación)** → convierte bytes en cadena

```
s = "Hola ñ"

b = s.encode("utf-8")
print(b)          # b'Hola \xc3\xb1'
print(type(b))    # <class 'bytes'>

s2 = b.decode("utf-8")
print(s2)         # 'Hola ñ'
```

Si usas una codificación incorrecta al decodificar, obtendrás errores o caracteres raros.

3.8 Errores comunes

Trabajando con cadenas es fácil cometer ciertos errores. Veamos los más típicos.

3.8.1 Índices fuera de rango

```
texto = "Hola"
# print(texto[10])  # IndexError: string index out of range
```

Solución: comprobar la longitud con `len()` antes de acceder.

3.8.2 Confundir tipos (str vs int/float)

```
edad = 30
# mensaje = "Tengo " + edad + " años"  # TypeError
```

Solución: convertir a cadena:

```
mensaje = "Tengo " + str(edad) + " años"
print(mensaje)
```

3.8.3 Inmutabilidad

Intentar modificar un carácter directamente:

```
s = "Hola"
# s[0] = "h"  # TypeError
```

Solución: crear una nueva cadena:

```
s = "Hola"
s = "h" + s[1:]
print(s)  # 'hola'
```

3.8.4 Búsquedas que no encuentran nada

```
texto = "Python"
pos = texto.find("z")
print(pos)  # -1
```

Si usas `index()` en lugar de `find()`:

```
# texto.index("z")  # ValueError
```

Solución: usar `find()` y comprobar si devuelve -1, o capturar la excepción de `index()`.

3.8.5 Problemas de codificación

Al leer o escribir archivos con caracteres especiales, si no se especifica la codificación correcta, pueden aparecer caracteres raros.

```
# Ejemplo conceptual:
# open("archivo.txt", "r", encoding="utf-8")
```

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Crea una cadena con tu nombre completo y muestra su longitud con `len()`.

Ejercicio 2
Pide al usuario una palabra y muestra su primer y último carácter usando indexación.

Ejercicio 3
Pide al usuario una palabra y muestra la palabra al revés usando slicing.

Ejercicio 4
Pide al usuario una frase y muestra cuántas letras tiene (sin contar espacios).

Ejercicio 5
Pide al usuario una frase y muestra la misma frase en mayúsculas, minúsculas y con la primera letra de cada palabra en mayúscula.

Ejercicio 6
Pide al usuario una frase y una palabra, y muestra cuántas veces aparece esa palabra en la frase.

Ejercicio 7
Pide al usuario una frase y reemplaza todas las apariciones de la palabra `"Python"` por `"Java"`.

Ejercicio 8
Pide al usuario una frase y muestra la subcadena formada por los 5 primeros caracteres y los 5 últimos.

Ejercicio 9
Pide al usuario una frase y muestra si contiene la palabra `"hola"` (sin distinguir mayúsculas/minúsculas).

Ejercicio 10
Pide al usuario una lista de palabras separadas por comas y conviértela en una lista de Python usando `split()`. Luego vuelve a unirlas con `";"` usando `join()`.

Ejercicio 11
Pide al usuario su nombre y su edad, y muestra un mensaje formateado usando f-strings.

Ejercicio 12
Crea una cadena multilínea que represente un pequeño poema o texto de varias líneas y muéstralo por pantalla.

Ejercicio 13
Pide al usuario una cadena y muestra solo los caracteres que están en las posiciones pares (0, 2, 4, …).

Ejercicio 14
Pide al usuario una cadena y comprueba si es un palíndromo (se lee igual al derecho y al revés), ignorando espacios y mayúsculas/minúsculas.

Ejercicio 15
Pide al usuario una frase y muestra una versión “limpia” sin espacios al principio ni al final, y sin dobles espacios en medio (reemplázalos por uno solo).

Ejercicio 16
Pide al usuario una frase y muestra el índice de la primera aparición de la letra `"a"`. Si no aparece, muestra un mensaje indicándolo.

Ejercicio 17
Pide al usuario una frase y codifícala en bytes usando UTF-8. Muestra los bytes resultantes y luego decodifícalos de nuevo a cadena.

Ejercicio 18
Pide al usuario una frase y muestra cuántas vocales (`a, e, i, o, u`) contiene, sin distinguir mayúsculas/minúsculas.

Ejercicio 19
Pide al usuario una frase y una subcadena, y muestra todas las posiciones (índices) en las que aparece esa subcadena.

Ejercicio 20
Pide al usuario una frase y muestra una nueva frase donde cada palabra empiece por mayúscula y el resto esté en minúscula (sin usar directamente `title()`, hazlo "a mano" con split, lower, upper, etc.).

Soluciones de los 20 ejercicios

Solución 1

```
nombre = "Tu Nombre Completo"  # o input("Introduce tu nombre completo: ")
longitud = len(nombre)
print("La longitud de tu nombre es:", longitud)
```

Solución 2

```
palabra = input("Introduce una palabra: ")

if len(palabra) > 0:
    primero = palabra[0]
    ultimo = palabra[-1]
    print("Primer carácter:", primero)
    print("Último carácter:", ultimo)
else:
    print("Has introducido una cadena vacía.")
```

Solución 3

```
palabra = input("Introduce una palabra: ")
```

```
al_reves = palabra[::-1]
print("Palabra al revés:", al_reves)
```

Solución 4

```
frase = input("Introduce una frase: ")

sin_espacios = frase.replace(" ", "")
longitud = len(sin_espacios)

print("Número de letras (sin espacios):", longitud)
```

Solución 5

```
frase = input("Introduce una frase: ")

mayus = frase.upper()
minus = frase.lower()
titulo = frase.title()

print("Mayúsculas:", mayus)
print("Minúsculas:", minus)
print("Título:", titulo)
```

Solución 6

```
frase = input("Introduce una frase: ")
palabra = input("Introduce una palabra a buscar: ")

# Para no distinguir mayúsculas/minúsculas:
frase_min = frase.lower()
palabra_min = palabra.lower()

veces = frase_min.count(palabra_min)
```

```
print(f"La palabra '{palabra}' aparece {veces} veces.")
```

Solución 7

```
frase = input("Introduce una frase: ")

nueva = frase.replace("Python", "Java")
print("Frase modificada:", nueva)
```

Solución 8

```
frase = input("Introduce una frase: ")

primeros_5 = frase[:5]
ultimos_5 = frase[-5:] if len(frase) >= 5 else frase

print("Primeros 5 caracteres:", primeros_5)
print("Últimos 5 caracteres:", ultimos_5)
```

Solución 9

```
frase = input("Introduce una frase: ")

frase_min = frase.lower()

if "hola" in frase_min:
    print("La frase contiene 'hola'.")
else:
    print("La frase NO contiene 'hola'.")
```

Solución 10

```
cadena = input("Introduce palabras separadas por comas: ")

lista = cadena.split(",")
print("Lista:", lista)
```

```
unidas = ";".join(lista)
print("Palabras unidas con ';':", unidas)
```

Solución 11

```
nombre = input("Introduce tu nombre: ")
edad = input("Introduce tu edad: ")

mensaje = f"Me llamo {nombre} y tengo {edad} años."
print(mensaje)
```

Solución 12

```
poema = """Este es un pequeño poema
escrito en varias líneas
para probar cadenas multilínea."""

print(poema)
```

Solución 13

```
cadena = input("Introduce una cadena: ")

pares = cadena[::2]  # caracteres en posiciones 0, 2, 4, ...

print("Caracteres en posiciones pares:", pares)
```

Solución 14

```
cadena = input("Introduce una cadena: ")

# Eliminar espacios y pasar a minúsculas
limpia = cadena.replace(" ", "").lower()

if limpia == limpia[::-1]:
```

```
    print("Es un palíndromo.")
else:
    print("No es un palíndromo.")
```

Solución 15

```
frase = input("Introduce una frase: ")

# Quitar espacios al principio y al final
frase = frase.strip()

# Reemplazar dobles espacios por uno solo repetidamente
while "  " in frase:
    frase = frase.replace("  ", " ")

print("Frase limpia:", frase)
```

Solución 16

```
frase = input("Introduce una frase: ")

indice = frase.find("a")

if indice == -1:
    print("La letra 'a' no aparece en la frase.")
else:
    print("La primera 'a' aparece en el índice:", indice)
```

Solución 17

```
frase = input("Introduce una frase: ")

bytes_utf8 = frase.encode("utf-8")
print("Bytes codificados:", bytes_utf8)
```

```
frase_decodificada = bytes_utf8.decode("utf-8")
print("Frase decodificada:", frase_decodificada)
```

Solución 18

```
frase = input("Introduce una frase: ")

frase_min = frase.lower()
vocales = "aeiou"
contador = 0

for c in frase_min:
    if c in vocales:
        contador += 1

print("Número de vocales:", contador)
```

Solución 19

```
frase = input("Introduce una frase: ")
sub = input("Introduce una subcadena a buscar: ")

inicio = 0
indices = []

while True:
    pos = frase.find(sub, inicio)
    if pos == -1:
        break
    indices.append(pos)
    inicio = pos + 1  # seguir buscando a partir del siguiente carácter

if indices:
```

```
    print("La subcadena aparece en los índices:", indices)
else:
    print("La subcadena no aparece en la frase.")
```

Solución 20

```
frase = input("Introduce una frase: ")

palabras = frase.split()
nuevas_palabras = []

for p in palabras:
    p_min = p.lower()
    if len(p_min) > 0:
        p_formateada = p_min[0].upper() + p_min[1:]
    else:
        p_formateada = p_min
    nuevas_palabras.append(p_formateada)

nueva_frase = " ".join(nuevas_palabras)
print("Frase con cada palabra capitalizada:", nueva_frase)
```

---

4. Estructuras condicionales en Python

Las **estructuras condicionales** son el mecanismo que permite que un programa **tome decisiones**. Hasta ahora, con variables y operadores, solo podíamos calcular cosas; con las condicionales podemos decir: “si pasa X, haz Y; si no, haz Z”.
Son la base de cualquier lógica: menús, validaciones, flujos de trabajo, juegos, sistemas de control, etc.

En esta lección vamos a ver, con mucha claridad y muchos ejemplos:

- **4.1 if**
- **4.2 if–else**
- **4.3 if–elif–else**
- **4.4 Condiciones múltiples**
- **4.5 Operadores lógicos**
- **4.6 Condiciones anidadas**
- **4.7 Buenas prácticas**
- **4.8 Errores frecuentes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

4.1 if

La sentencia **if** permite ejecutar un bloque de código **solo si** se cumple una condición (es decir, si la expresión booleana es `True`).

Sintaxis básica:

```
if condicion:
    # bloque de código
    # se ejecuta solo si condicion es True
```

La **condición** suele ser una expresión que devuelve `True` o `False`, normalmente usando operadores relacionales (`==`, `!=`, `<`, `>`, `<=`, `>=`) y lógicos (`and`, `or`, `not`).

Ejemplo 1: comprobar si un número es positivo

```
numero = int(input("Introduce un número entero: "))


if numero > 0:
    print("El número es positivo.")
```

Si `numero > 0` es `True`, se ejecuta el `print`. Si es `False`, se salta el bloque.

Ejemplo 2: comprobar si el usuario es mayor de edad

```
edad = int(input("Introduce tu edad: "))
```

```
if edad >= 18:
    print("Eres mayor de edad.")
```

Si la condición no se cumple, simplemente no pasa nada (no hay mensaje).

4.2 if–else

Muchas veces queremos hacer **una cosa si la condición se cumple** y **otra cosa si no se cumple**. Para eso usamos **if–else**.

Sintaxis:

```
if condicion:
    # bloque si condicion es True
else:
    # bloque si condicion es False
```

Ejemplo 3: mayor de edad o menor de edad

```
edad = int(input("Introduce tu edad: "))

if edad >= 18:
    print("Eres mayor de edad.")
else:
    print("Eres menor de edad.")
```

Ejemplo 4: número par o impar

```
numero = int(input("Introduce un número entero: "))

if numero % 2 == 0:
    print("El número es par.")
else:
    print("El número es impar.")
```

4.3 if–elif–else

Cuando tenemos **más de dos casos**, podemos encadenar condiciones usando **elif** (abreviatura de "else if").

Sintaxis:

```
if condicion1:
```

```
    # bloque si condicion1 es True
elif condicion2:
    # bloque si condicion2 es True
elif condicion3:
    # bloque si condicion3 es True
else:
    # bloque si ninguna condición anterior es True
```

Python evalúa las condiciones **en orden** y ejecuta **solo el primer bloque** cuya condición sea `True`.

Ejemplo 5: clasificación de notas

```
nota = float(input("Introduce tu nota (0-10): "))

if nota < 5:
    print("Suspenso")
elif nota < 7:
    print("Aprobado")
elif nota < 9:
    print("Notable")
else:
    print("Sobresaliente")
```

Observa que no hace falta escribir `nota >= 5` en el segundo caso, porque si el programa llega ahí es que ya sabemos que `nota >= 5`.

Ejemplo 6: menú simple

```
print("Menú:")
print("1. Sumar")
print("2. Restar")
print("3. Multiplicar")
print("4. Dividir")

opcion = input("Elige una opción (1-4): ")

if opcion == "1":
    print("Has elegido sumar.")
```

```
elif opcion == "2":
    print("Has elegido restar.")
elif opcion == "3":
    print("Has elegido multiplicar.")
elif opcion == "4":
    print("Has elegido dividir.")
else:
    print("Opción no válida.")
```

4.4 Condiciones múltiples

Una **condición** puede ser tan simple como `x > 0`, pero también puede ser más compleja, combinando varias comparaciones con operadores lógicos.

Ejemplos típicos:

- Comprobar si un valor está en un **rango**: `18 <= edad <= 65`
- Comprobar varias cosas a la vez: `edad >= 18 and tiene_carnet`

Ejemplo 7: rango de edad

```
edad = int(input("Introduce tu edad: "))

if 18 <= edad <= 65:
    print("Edad en rango laboral típico.")
else:
    print("Edad fuera del rango 18-65.")
```

Ejemplo 8: validación de usuario y contraseña

```
usuario_correcto = "admin"
contrasena_correcta = "1234"

usuario = input("Usuario: ")
contrasena = input("Contraseña: ")

if usuario == usuario_correcto and contrasena ==
contrasena_correcta:
    print("Acceso permitido.")
else:
```

```
    print("Usuario o contraseña incorrectos.")
```

4.5 Operadores lógicos

Los **operadores lógicos** permiten combinar condiciones:

- **and** → verdadero si **ambas** condiciones son verdaderas
- **or** → verdadero si **al menos una** condición es verdadera
- **not** → niega el valor lógico (True → False, False → True)

Ejemplo 9: and

```
edad = int(input("Edad: "))
tiene_carnet = input("¿Tiene carnet? (s/n): ").lower() == "s"

if edad >= 18 and tiene_carnet:
    print("Puede conducir.")
else:
    print("No puede conducir.")
```

Ejemplo 10: or

```
dia = input("Introduce el día de la semana: ").lower()

if dia == "sabado" or dia == "domingo":
    print("Es fin de semana.")
else:
    print("Es día laborable.")
```

Ejemplo 11: not

```
llueve = input("¿Está lloviendo? (s/n): ").lower() == "s"

if not llueve:
    print("Puedes salir sin paraguas.")
else:
    print("Mejor lleva paraguas.")
```

4.6 Condiciones anidadas

Las **condiciones anidadas** aparecen cuando dentro de un bloque `if` (o `else`) ponemos **otro if**. Esto permite construir decisiones más complejas.

Ejemplo 12: descuento según edad y socio

```
edad = int(input("Introduce tu edad: "))
es_socio = input("¿Eres socio? (s/n): ").lower() == "s"

if edad < 18:
    if es_socio:
        print("Descuento del 30% (menor de edad y socio).")
    else:
        print("Descuento del 10% (menor de edad).")
else:
    if es_socio:
        print("Descuento del 20% (adulto socio).")
    else:
        print("Sin descuento.")
```

Aquí hay **if dentro de if**, formando una estructura de árbol de decisiones.

Ejemplo 13: validación de entrada

```
texto = input("Introduce un número entero: ")

if texto.isdigit():
    numero = int(texto)
    if numero > 0:
        print("Has introducido un entero positivo.")
    else:
        print("Has introducido un entero no positivo.")
else:
    print("Eso no es un número entero válido.")
```

4.7 Buenas prácticas

Al usar condicionales, conviene seguir algunas **buenas prácticas** para que el código sea más claro y menos propenso a errores.

Buena práctica 1: condiciones claras y legibles

Evita condiciones demasiado largas y difíciles de leer. Si es necesario, divide la lógica en variables intermedias:

```
edad = int(input("Edad: "))
tiene_carnet = input("¿Tiene carnet? (s/n): ").lower() == "s"
ha_bebido = input("¿Ha bebido alcohol? (s/n): ").lower() == "s"

condicion_edad = edad >= 18
condicion_carnet = tiene_carnet
condicion_alcohol = not ha_bebido

if condicion_edad and condicion_carnet and condicion_alcohol:
    print("Puede conducir.")
else:
    print("No puede conducir.")
```

Buena práctica 2: evitar duplicar lógica

En lugar de repetir la misma condición en varios sitios, intenta agruparla.

Buena práctica 3: usar `elif` en lugar de múltiples `if` independientes

Si los casos son **mutuamente excluyentes**, usa `elif`:

```
nota = float(input("Nota: "))

if nota < 5:
    print("Suspenso")
elif nota < 7:
    print("Aprobado")
elif nota < 9:
    print("Notable")
else:
    print("Sobresaliente")
```

Si usaras varios `if` separados, podrías imprimir más de un mensaje.

Buena práctica 4: usar `is None` para comprobar valores nulos

```
valor = None

if valor is None:
    print("No hay valor.")
```

```
else:
    print("Hay un valor.")
```

4.8 Errores frecuentes

Veamos algunos errores típicos al trabajar con condicionales.

Error 1: usar = en lugar de ==

```
x = 5
# if x = 5:  # ERROR de sintaxis
#     ...
```

= es asignación, == es comparación.

Error 2: olvidar los dos puntos :

```
# if x > 0
#     print("Positivo")
```

Debe ser:

```
if x > 0:
    print("Positivo")
```

Error 3: mala indentación

La indentación (espacios al principio de la línea) define los bloques:

```
if x > 0:
print("Positivo")  # Esto da error de indentación
```

Debe ser:

```
if x > 0:
    print("Positivo")
```

Error 4: condiciones que nunca se cumplen

```
edad = 20

if edad < 18:
    print("Menor")
elif edad < 10:
    print("Niño")
```

La segunda condición `edad < 10` nunca se ejecutará, porque si `edad < 10` también es `< 18`, ya habría entrado en el primer bloque.

Error 5: confundir `and` y `or`

```
edad = 20


if edad < 18 or edad > 65:
    print("Fuera del rango 18-65")
```

Si querías “dentro del rango”, deberías usar `and` y cambiar la lógica.

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Pide un número entero y muestra si es positivo, negativo o cero.

Ejercicio 2
Pide una edad y muestra “Mayor de edad” si es ≥ 18 y “Menor de edad” en caso contrario.

Ejercicio 3
Pide una nota (0–10) y muestra “Suspenso”, “Aprobado”, “Notable” o “Sobresaliente” usando `if-elif-else`.

Ejercicio 4
Pide un número entero y muestra si es par o impar.

Ejercicio 5
Pide un año y muestra si es bisiesto o no. Un año es bisiesto si es divisible por 4 y no por 100, salvo que también sea divisible por 400.

Ejercicio 6
Pide al usuario un nombre de usuario y una contraseña. Si el usuario es `"admin"` y la contraseña `"1234"`, muestra “Acceso permitido”; en caso contrario, “Acceso denegado”.

Ejercicio 7
Pide al usuario un número entero y muestra si está en el rango [1, 100]. Usa una condición múltiple.

Ejercicio 8
Pide al usuario un día de la semana y muestra si es laborable (lunes–viernes) o fin de semana (sábado–domingo).

Ejercicio 9
Pide al usuario su edad y si tiene carnet (s/n). Muestra si puede conducir (edad ≥ 18 y carnet “s”).

Ejercicio 10
Pide al usuario un número entero y muestra si es múltiplo de 3 y de 5 a la vez, solo de 3, solo de 5 o de ninguno.

Ejercicio 11
Pide al usuario una cadena y muestra si está vacía o no (usa `len()` y un `if`).

Ejercicio 12
Pide al usuario una cadena y muestra si contiene la letra `"a"` (sin distinguir mayúsculas/minúsculas).

Ejercicio 13
Pide al usuario una nota (0–10). Si la nota es menor que 0 o mayor que 10, muestra “Nota inválida”; en caso contrario, clasifícala como en el ejercicio 3.

Ejercicio 14
Pide al usuario tres números y muestra cuál es el mayor (usa condicionales).

Ejercicio 15
Pide al usuario un número entero y muestra un mensaje diferente según sea menor que 10, entre 10 y 20, entre 21 y 50, o mayor que 50.

Ejercicio 16
Pide al usuario un número entero y muestra si es positivo y par, positivo e impar, negativo y par, negativo e impar, o cero.

Ejercicio 17
Pide al usuario una contraseña y comprueba si cumple estas reglas: longitud ≥ 8, contiene al menos una letra y al menos un dígito. Muestra mensajes adecuados.

Ejercicio 18
Pide al usuario una temperatura en grados Celsius y muestra si hace frío (< 10), templado (10–25) o calor (> 25).

Ejercicio 19
Pide al usuario un número entero y muestra si está en el rango [1, 10] o en el rango [20, 30], o en ninguno de los dos.

Ejercicio 20
Pide al usuario un número entero y muestra si es primo o no (un número primo solo es divisible por 1 y por sí mismo). Usa condicionales y un pequeño bucle.

Soluciones de los 20 ejercicios

Solución 1

```
n = int(input("Introduce un número entero: "))


if n > 0:
    print("El número es positivo.")
elif n < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")
```

Solución 2

```
edad = int(input("Introduce tu edad: "))

if edad >= 18:
    print("Mayor de edad.")
else:
    print("Menor de edad.")
```

Solución 3

```
nota = float(input("Introduce tu nota (0-10): "))

if nota < 5:
    print("Suspenso")
elif nota < 7:
    print("Aprobado")
elif nota < 9:
    print("Notable")
else:
    print("Sobresaliente")
```

Solución 4

```
n = int(input("Introduce un número entero: "))

if n % 2 == 0:
    print("El número es par.")
else:
    print("El número es impar.")
```

Solución 5

```
anio = int(input("Introduce un año: "))
```

```
es_bisiesto = (anio % 4 == 0 and anio % 100 != 0) or (anio % 400
== 0)

if es_bisiesto:
    print("El año es bisiesto.")
else:
    print("El año no es bisiesto.")
```

Solución 6

```
usuario = input("Usuario: ")
contrasena = input("Contraseña: ")

if usuario == "admin" and contrasena == "1234":
    print("Acceso permitido.")
else:
    print("Acceso denegado.")
```

Solución 7

```
n = int(input("Introduce un número entero: "))

if 1 <= n <= 100:
    print("El número está en el rango [1, 100].")
else:
    print("El número está fuera del rango.")
```

Solución 8

```
dia = input("Introduce un día de la semana: ").lower()

if dia in ["lunes", "martes", "miercoles", "miércoles", "jueves",
"viernes"]:
    print("Es día laborable.")
elif dia in ["sabado", "sábado", "domingo"]:
    print("Es fin de semana.")
```

```
else:
    print("Día no reconocido.")
```

Solución 9

```
edad = int(input("Edad: "))
tiene_carnet = input("¿Tiene carnet? (s/n): ").lower() == "s"

if edad >= 18 and tiene_carnet:
    print("Puede conducir.")
else:
    print("No puede conducir.")
```

Solución 10

```
n = int(input("Introduce un número entero: "))

multiplo_3 = (n % 3 == 0)
multiplo_5 = (n % 5 == 0)

if multiplo_3 and multiplo_5:
    print("Es múltiplo de 3 y de 5.")
elif multiplo_3:
    print("Es múltiplo solo de 3.")
elif multiplo_5:
    print("Es múltiplo solo de 5.")
else:
    print("No es múltiplo de 3 ni de 5.")
```

Solución 11

```
cadena = input("Introduce una cadena: ")

if len(cadena) == 0:
    print("La cadena está vacía.")
```

```
else:
    print("La cadena NO está vacía.")
```

Solución 12

```
cadena = input("Introduce una cadena: ")

cadena_min = cadena.lower()

if "a" in cadena_min:
    print("La cadena contiene la letra 'a'.")
else:
    print("La cadena NO contiene la letra 'a'.")
```

Solución 13

```
nota = float(input("Introduce tu nota (0-10): "))

if nota < 0 or nota > 10:
    print("Nota inválida.")
else:
    if nota < 5:
        print("Suspenso")
    elif nota < 7:
        print("Aprobado")
    elif nota < 9:
        print("Notable")
    else:
        print("Sobresaliente")
```

Solución 14

```
a = float(input("Introduce el primer número: "))
b = float(input("Introduce el segundo número: "))
c = float(input("Introduce el tercer número: "))
```

```
mayor = a

if b > mayor:
    mayor = b
if c > mayor:
    mayor = c

print("El mayor es:", mayor)
```

Solución 15

```
n = int(input("Introduce un número entero: "))

if n < 10:
    print("Menor que 10.")
elif 10 <= n <= 20:
    print("Entre 10 y 20.")
elif 21 <= n <= 50:
    print("Entre 21 y 50.")
else:
    print("Mayor que 50.")
```

Solución 16

```
n = int(input("Introduce un número entero: "))

if n == 0:
    print("El número es cero.")
else:
    es_positivo = n > 0
    es_par = (n % 2 == 0)

    if es_positivo and es_par:
```

```
        print("El número es positivo y par.")
    elif es_positivo and not es_par:
        print("El número es positivo e impar.")
    elif not es_positivo and es_par:
        print("El número es negativo y par.")
    else:
        print("El número es negativo e impar.")
```

Solución 17

```
contrasena = input("Introduce una contraseña: ")

tiene_longitud = len(contrasena) >= 8
tiene_letra = any(c.isalpha() for c in contrasena)
tiene_digito = any(c.isdigit() for c in contrasena)

if tiene_longitud and tiene_letra and tiene_digito:
    print("Contraseña válida.")
else:
    print("Contraseña inválida. Debe tener al menos 8 caracteres, letras y dígitos.")
```

Solución 18

```
temp = float(input("Introduce la temperatura en grados Celsius: "))

if temp < 10:
    print("Hace frío.")
elif temp <= 25:
    print("Temperatura templada.")
else:
    print("Hace calor.")
```

Solución 19

```
n = int(input("Introduce un número entero: "))

if 1 <= n <= 10:
    print("El número está en el rango [1, 10].")
elif 20 <= n <= 30:
    print("El número está en el rango [20, 30].")
else:
    print("El número no está en ninguno de los dos rangos.")
```

Solución 20

```
n = int(input("Introduce un número entero: "))

if n <= 1:
    es_primo = False
else:
    es_primo = True
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            es_primo = False
            break

if es_primo:
    print("El número es primo.")
else:
    print("El número no es primo.")
```

---

5. Estructuras de repetición: bucles en Python

Las **estructuras de repetición** (bucles) permiten que un programa ejecute **varias veces** un mismo bloque de código.
Son esenciales para:

- **Recorrer listas, cadenas, diccionarios…**
- **Procesar datos** elemento a elemento.
- **Realizar cálculos repetitivos** (sumas, productos, estadísticas).
- **Simular procesos** (juegos, animaciones, simulaciones numéricas).
- **Automatizar tareas** (leer muchos archivos, generar informes, etc.).

En esta lección veremos con mucha claridad:

- **5.1 Bucle for**
- **5.2 Bucle while**
- **5.3 break, continue, pass**
- **5.4 Bucles anidados**
- **5.5 Iteración sobre colecciones**
- **5.6 List comprehensions**
- **5.7 Errores típicos**

Al final tendrás **20 ejercicios** con su **solución detallada**.

5.1 Bucle for

El **bucle for** se usa para **recorrer secuencias**: listas, cadenas, rangos de números, etc.
La idea es: “para cada elemento de esta colección, haz algo”.

Sintaxis básica:

```
for variable in secuencia:
    # bloque de código
```

- **variable**: toma, en cada iteración, el valor de un elemento de la secuencia.
- **secuencia**: puede ser una lista, una cadena, un rango (`range()`), etc.

5.1.1 Ejemplo simple con lista

```
numeros = [10, 20, 30, 40]


for n in numeros:
    print("Número:", n)
```

Salida:

```
Número: 10
Número: 20
Número: 30
```

```
Número: 40
```

5.1.2 Ejemplo con cadena

```
texto = "Python"

for caracter in texto:
    print(caracter)
```

Salida:

```
P
y
t
h
o
n
```

5.1.3 Ejemplo con range()

`range()` genera una secuencia de números enteros.

```
for i in range(5):
    print("i vale:", i)
```

Salida:

```
i vale: 0
i vale: 1
i vale: 2
i vale: 3
i vale: 4
```

Formas de range():

- `range(fin)` → desde 0 hasta `fin - 1`
- `range(inicio, fin)` → desde `inicio` hasta `fin - 1`
- `range(inicio, fin, paso)` → desde `inicio` hasta `fin - 1`, saltando de `paso` en `paso`

Ejemplos:

```
for i in range(1, 6):
    print(i)  # 1, 2, 3, 4, 5

for i in range(0, 10, 2):
```

```
    print(i)  # 0, 2, 4, 6, 8
```

5.2 Bucle while

El **bucle while** repite un bloque de código **mientras** se cumpla una condición booleana.

Sintaxis básica:

```
while condicion:
    # bloque de código
```

- Mientras `condicion` sea `True`, el bloque se ejecuta.
- Cuando `condicion` pasa a ser `False`, el bucle termina.

5.2.1 Ejemplo simple

```
contador = 1

while contador <= 5:
    print("Contador:", contador)
    contador += 1  # incrementamos
```

Salida:

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
```

5.2.2 Ejemplo con lectura de usuario

```
texto = ""

while texto != "salir":
    texto = input("Escribe algo (o 'salir' para terminar): ")
    print("Has escrito:", texto)

print("Programa terminado.")
```

Este bucle se repite hasta que el usuario escribe `"salir"`.

5.2.3 Diferencias clave entre for y while

- **for**: se usa cuando sabes **cuántas veces** quieres repetir (por ejemplo, recorrer una lista, un rango).
- **while**: se usa cuando quieres repetir **mientras se cumpla una condición**, y no necesariamente sabes cuántas iteraciones habrá (por ejemplo, leer hasta que el usuario escriba “salir”).

5.3 break, continue, pass

Dentro de los bucles podemos usar algunas **sentencias especiales** para controlar el flujo:

- **break** → sale del bucle inmediatamente.
- **continue** → salta al siguiente ciclo (no ejecuta el resto del bloque en esa iteración).
- **pass** → no hace nada; se usa como “relleno” cuando necesitamos una sentencia pero aún no queremos escribir código.

5.3.1 break

```
for i in range(1, 11):
    if i == 5:
        break
    print(i)
```

Salida:

```
1
2
3
4
```

Cuando `i == 5`, se ejecuta `break` y el bucle termina.

5.3.2 continue

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Salida:

```
1
2
4
5
```

Cuando `i == 3`, se ejecuta `continue` y se **salta** el `print(i)` de esa iteración.

5.3.3 pass

```
for i in range(5):
    if i == 2:
        pass  # aquí podríamos poner código en el futuro
    print("i =", i)
```

`pass` no hace nada, pero permite que el código sea sintácticamente correcto.

También se usa en estructuras que aún no hemos implementado:

```
def funcion_por_definir():
    pass  # ya la implementaremos más tarde
```

5.4 Bucles anidados

Un **bucle anidado** es un bucle dentro de otro bucle.
Se usan mucho para trabajar con **tablas**, **matrices**, **combinaciones de valores**, etc.

5.4.1 Ejemplo: tabla de multiplicar

```
for i in range(1, 4):      # filas
    for j in range(1, 4):  # columnas
        producto = i * j
        print(f"{i} x {j} = {producto}")
    print("---")
```

Salida:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
---
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
---
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

5.4.2 Ejemplo: recorrer una matriz (lista de listas)

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print()
```

Salida:

```
1 2 3
4 5 6
7 8 9
```

5.5 Iteración sobre colecciones

Las colecciones más típicas en Python son:

- **listas** (`list`)
- **tuplas** (`tuple`)
- **diccionarios** (`dict`)
- **conjuntos** (`set`)
- **cadenas** (`str`)

El bucle `for` se usa para **iterar** sobre ellas.

5.5.1 Listas

```
frutas = ["manzana", "pera", "plátano"]

for fruta in frutas:
    print("Me gusta la", fruta)
```

5.5.2 Tuplas

```
punto = (3, 4)
```

```
for coordenada in punto:
    print("Coordenada:", coordenada)
```

5.5.3 Diccionarios

```
persona = {
    "nombre": "Luis",
    "edad": 30,
    "ciudad": "Zaragoza"
}

# Iterar claves
for clave in persona:
    print("Clave:", clave, "Valor:", persona[clave])

# Iterar items (pares clave-valor)
for clave, valor in persona.items():
    print(clave, "=>", valor)
```

5.5.4 Conjuntos (set)

```
numeros = {1, 2, 3, 2, 1}

for n in numeros:
    print(n)
```

Los conjuntos no tienen orden definido, así que la salida puede variar.

5.5.5 Enumerar elementos con índice: enumerate()

```
frutas = ["manzana", "pera", "plátano"]

for indice, fruta in enumerate(frutas):
    print(indice, "=>", fruta)
```

Salida:

```
0 => manzana
1 => pera
2 => plátano
```

5.6 List comprehensions

Las **list comprehensions** son una forma **compacta y elegante** de crear listas usando bucles y condiciones en una sola línea.

Sintaxis básica:

```
nueva_lista = [expresion for elemento in secuencia if condicion]
```

- Se recorre `secuencia`.
- Para cada `elemento` que cumple `condicion` (si se incluye), se evalúa `expresion`.
- El resultado se añade a `nueva_lista`.

5.6.1 Ejemplo simple: cuadrados de números

```
cuadrados = [x**2 for x in range(1, 6)]
print(cuadrados)  # [1, 4, 9, 16, 25]
```

5.6.2 Ejemplo con condición: pares

```
pares = [x for x in range(1, 11) if x % 2 == 0]
print(pares)  # [2, 4, 6, 8, 10]
```

5.6.3 Ejemplo: procesar cadenas

```
palabras = ["Python", "es", "genial"]

longitudes = [len(p) for p in palabras]
print(longitudes)  # [6, 2, 6]
```

5.6.4 Ejemplo: transformar y filtrar

```
numeros = [1, -2, 3, -4, 5]

positivos_cuadrados = [n**2 for n in numeros if n > 0]
print(positivos_cuadrados)  # [1, 9, 25]
```

Las list comprehensions son muy usadas en Python por su **claridad** y **concisión**.

5.7 Errores típicos

Trabajando con bucles es fácil cometer ciertos errores. Veamos los más frecuentes.

5.7.1 Bucle infinito con while

```
contador = 1

while contador <= 5:
    print(contador)
```

```
    # ¡Olvidamos incrementar contador!
```

Este bucle nunca termina, porque `contador` siempre vale 1.

Solución: asegurarse de que la condición pueda llegar a ser `False`:

```
contador = 1

while contador <= 5:
    print(contador)
    contador += 1
```

5.7.2 Modificar la lista mientras se itera

```
numeros = [1, 2, 3, 4, 5]

for n in numeros:
    if n % 2 == 0:
        numeros.remove(n)
```

Esto puede producir resultados inesperados, porque la lista se modifica mientras se recorre.

Solución: crear una nueva lista o iterar sobre una copia:

```
numeros = [1, 2, 3, 4, 5]
nueva = []

for n in numeros:
    if n % 2 != 0:
        nueva.append(n)

print(nueva)  # [1, 3, 5]
```

5.7.3 Usar mal range()

```
for i in range(1, 5):
    print(i)
```

Salida: 1, 2, 3, 4 (no incluye el 5).
A veces se olvida que el límite superior es **exclusivo**.

5.7.4 Olvidar break en bucles de búsqueda

```
numeros = [1, 2, 3, 4, 5]
buscado = 3
```

```
encontrado = False


for n in numeros:
    if n == buscado:
        encontrado = True
        # falta break


print("Encontrado:", encontrado)
```

Funciona, pero recorre toda la lista aunque ya haya encontrado el elemento.
Mejor:

```
for n in numeros:
    if n == buscado:
        encontrado = True
        break
```

5.7.5 Confundir for y while

Usar `while` cuando sería más claro usar `for`:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Se puede escribir más claro con `for`:

```
for i in range(5):
    print(i)
```

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Usa un bucle `for` para mostrar los números del 1 al 10.

Ejercicio 2
Usa un bucle `for` para mostrar los números pares del 2 al 20.

Ejercicio 3
Usa un bucle `for` y `range()` para mostrar la tabla de multiplicar del 5 (del 5×1 al 5×10).

Ejercicio 4
Pide al usuario una palabra y muestra cada carácter en una línea usando un `for`.

Ejercicio 5
Pide al usuario 5 números enteros y calcula su suma usando un `for`.

Ejercicio 6
Usa un bucle `while` para contar desde 10 hasta 1 (en orden descendente).

Ejercicio 7
Pide al usuario una contraseña y repite la petición con un `while` hasta que escriba `"secreto"`.

Ejercicio 8
Usa un `while` para sumar números introducidos por el usuario hasta que introduzca 0. Muestra la suma final.

Ejercicio 9
Usa un `for` para recorrer una lista de nombres y mostrar solo los que tienen más de 4 letras.

Ejercicio 10
Usa un `for` y `break` para buscar un número en una lista. Si lo encuentras, muestra “Encontrado” y termina el bucle.

Ejercicio 11
Usa un `for` y `continue` para mostrar los números del 1 al 10 excepto el 5 y el 7.

Ejercicio 12
Crea una lista de 10 números aleatorios entre 1 y 100 y muestra solo los que son mayores que 50 usando un `for`.

Ejercicio 13
Usa bucles anidados para mostrar todas las combinaciones de dos números entre 1 y 3 (por ejemplo, (1,1), (1,2), …).

Ejercicio 14
Usa bucles anidados para mostrar una “cuadrícula” de 3×3 con asteriscos (`*`).

Ejercicio 15
Usa una list comprehension para crear una lista con los cuadrados de los números del 1 al 10.

Ejercicio 16
Usa una list comprehension para crear una lista con los números pares del 1 al 20.

Ejercicio 17
Usa una list comprehension para crear una lista con las longitudes de las palabras de una lista dada.

Ejercicio 18
Pide al usuario una frase y usa un `for` para contar cuántas veces aparece la letra `"a"` (sin distinguir mayúsculas/minúsculas).

Ejercicio 19
Usa un `while` para pedir números al usuario y contar cuántos son positivos. El bucle termina cuando el usuario introduce un número negativo.

Ejercicio 20
Usa un `for` para recorrer una lista de números y crear otra lista con solo los números mayores que 10, sin usar list comprehension (usa `append`).

Soluciones de los 20 ejercicios

Solución 1

```
for i in range(1, 11):
    print(i)
```

Solución 2

```
for i in range(2, 21, 2):
    print(i)
```

Solución 3

```
for i in range(1, 11):
    print(f"5 x {i} = {5 * i}")
```

Solución 4

```
palabra = input("Introduce una palabra: ")

for c in palabra:
    print(c)
```

Solución 5

```
suma = 0

for i in range(5):
    n = int(input(f"Introduce el número {i+1}: "))
    suma += n

print("La suma es:", suma)
```

Solución 6

```
contador = 10

while contador >= 1:
    print(contador)
    contador -= 1
```

Solución 7

```
contrasena = ""

while contrasena != "secreto":
    contrasena = input("Introduce la contraseña (secreto): ")

print("Contraseña correcta.")
```

Solución 8

```
suma = 0
n = None

while n != 0:
    n = int(input("Introduce un número (0 para terminar): "))
    suma += n

print("La suma total es:", suma)
```

(Si no quieres sumar el 0 final, puedes sumar solo si `n != 0`.)

Solución 9

```
nombres = ["Ana", "Luis", "María", "Pedro", "Juan", "Beatriz"]

for nombre in nombres:
    if len(nombre) > 4:
        print(nombre)
```

Solución 10

```
numeros = [3, 8, 15, 7, 20]
buscado = int(input("Introduce el número a buscar: "))

encontrado = False

for n in numeros:
    if n == buscado:
        encontrado = True
        print("Encontrado.")
        break

if not encontrado:
    print("No se encontró el número.")
```

Solución 11

```
for i in range(1, 11):
    if i == 5 or i == 7:
        continue
    print(i)
```

Solución 12

```
import random

numeros = [random.randint(1, 100) for _ in range(10)]

print("Lista completa:", numeros)
print("Mayores que 50:")

for n in numeros:
    if n > 50:
```

```
        print(n)
```

Solución 13

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"({i}, {j})")
```

Solución 14

```
for i in range(3):
    for j in range(3):
        print("*", end=" ")
    print()
```

Salida:

```
* * *
* * *
* * *
```

Solución 15

```
cuadrados = [x**2 for x in range(1, 11)]
print(cuadrados)
```

Solución 16

```
pares = [x for x in range(1, 21) if x % 2 == 0]
print(pares)
```

Solución 17

```
palabras = ["Python", "es", "genial", "y", "potente"]

longitudes = [len(p) for p in palabras]
print(longitudes)
```

Solución 18

```
frase = input("Introduce una frase: ")

frase_min = frase.lower()
contador = 0

for c in frase_min:
    if c == "a":
        contador += 1

print("Número de 'a':", contador)
```

Solución 19

```
contador_positivos = 0

while True:
    n = int(input("Introduce un número (negativo para terminar): 
"))
    if n < 0:
        break
    if n > 0:
        contador_positivos += 1

print("Cantidad de números positivos introducidos:", 
contador_positivos)
```

Solución 20

```
numeros = [3, 12, 7, 25, 10, 18]
mayores_10 = []

for n in numeros:
    if n > 10:
        mayores_10.append(n)
```

```
print("Números mayores que 10:", mayores_10)
```

---

6. Listas en Python

Las **listas** son una de las estructuras de datos más importantes de Python.
Con ellas podemos almacenar **colecciones ordenadas y modificables** de elementos: números, cadenas, otras listas, objetos, mezclas de tipos, etc.
Casi cualquier programa real en Python usa listas de alguna forma: para manejar datos, resultados de consultas, colecciones de objetos, filas de un fichero, etc.

En esta lección veremos, con teoría muy detallada y muchos ejemplos:

- **6.1 Creación y características**
- **6.2 Indexación y slicing**
- **6.3 Métodos fundamentales**
- **6.4 Ordenación**
- **6.5 Copias superficiales y profundas**
- **6.6 Listas anidadas**
- **6.7 Errores comunes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

6.1 Creación y características

Una **lista** es una colección **ordenada**, **mutable** (se puede modificar) y que puede contener **elementos de cualquier tipo**.

Se define usando **corchetes** `[]` y separando los elementos con comas.

6.1.1 Creación básica de listas

```
vacia = []                      # lista vacía
numeros = [1, 2, 3, 4, 5]       # lista de enteros
nombres = ["Ana", "Luis", "Eva"]  # lista de cadenas
mixta = [1, "hola", 3.14, True] # lista con tipos mezclados


print(vacia)
print(numeros)
print(nombres)
print(mixta)
```

6.1.2 Longitud de una lista

La función **len()** devuelve el número de elementos de la lista:

```
numeros = [10, 20, 30]
print(len(numeros))  # 3
```

6.1.3 Listas como secuencias

Las listas son **secuencias**, igual que las cadenas:

- Tienen **índices** (0, 1, 2, …).
- Se pueden recorrer con **for**.
- Se pueden usar con **slicing**.

```
frutas = ["manzana", "pera", "plátano"]


for fruta in frutas:
    print("Me gusta la", fruta)
```

6.1.4 Mutabilidad

Una característica clave: las listas son **mutables**, es decir, podemos cambiar sus elementos:

```
numeros = [10, 20, 30]
numeros[1] = 99
print(numeros)  # [10, 99, 30]
```

Esto las diferencia de las **tuplas**, que son inmutables.

6.2 Indexación y slicing

Como las listas son secuencias, podemos acceder a sus elementos por **índice** y obtener **sublistas** con slicing.

6.2.1 Indexación

Los índices empiezan en **0**:

```
letras = ["a", "b", "c", "d", "e"]


print(letras[0])  # 'a'
print(letras[1])  # 'b'
print(letras[4])  # 'e'
```

También podemos usar índices **negativos**, que cuentan desde el final:

```
print(letras[-1])  # 'e'
print(letras[-2])  # 'd'
print(letras[-5])  # 'a'
```

Si intentamos acceder a un índice fuera de rango, obtenemos un **IndexError**:

```
# print(letras[10])  # IndexError
```

6.2.2 Slicing (rebanado)

El slicing permite obtener **sublistas**:

```
sublista = letras[inicio:fin:paso]
```

- **inicio**: índice inicial (incluido).
- **fin**: índice final (excluido).
- **paso**: salto entre índices (opcional).

Ejemplos:

```
letras = ["a", "b", "c", "d", "e", "f"]


print(letras[0:3])   # ['a', 'b', 'c']
print(letras[2:5])   # ['c', 'd', 'e']
print(letras[:4])    # ['a', 'b', 'c', 'd']
print(letras[3:])    # ['d', 'e', 'f']
print(letras[:])     # copia completa
```

Con paso:

```
print(letras[0:6:2])  # ['a', 'c', 'e']
print(letras[::2])    # ['a', 'c', 'e']
print(letras[::-1])   # ['f', 'e', 'd', 'c', 'b', 'a'] (lista invertida)
```

6.2.3 Modificar por slicing

Podemos **asignar** a un slice:

```
numeros = [1, 2, 3, 4, 5]
numeros[1:4] = [20, 30, 40]
print(numeros)  # [1, 20, 30, 40, 5]
```

También podemos **insertar** o **eliminar** elementos:

```
numeros = [1, 2, 3, 4, 5]
numeros[1:1] = [100, 200]  # inserta en la posición 1
print(numeros)  # [1, 100, 200, 2, 3, 4, 5]


numeros[2:5] = []  # elimina elementos
print(numeros)
```

6.3 Métodos fundamentales

Las listas tienen muchos **métodos** muy útiles. Vamos a ver los más importantes:

- **append(x)** → añade un elemento al final.
- **insert(i, x)** → inserta un elemento en la posición `i`.
- **extend(iterable)** → añade varios elementos (concatena).
- **remove(x)** → elimina la primera aparición de `x`.
- **pop(i)** → elimina y devuelve el elemento en la posición `i` (por defecto, el último).
- **clear()** → elimina todos los elementos.
- **index(x)** → devuelve el índice de la primera aparición de `x`.
- **count(x)** → cuenta cuántas veces aparece `x`.
- **sort()** → ordena la lista.
- **reverse()** → invierte el orden de la lista.

6.3.1 append()

```
numeros = [1, 2, 3]
numeros.append(4)
numeros.append(5)
print(numeros)  # [1, 2, 3, 4, 5]
```

6.3.2 insert()

```
numeros = [1, 2, 3]
numeros.insert(1, 99)  # inserta 99 en la posición 1
print(numeros)  # [1, 99, 2, 3]
```

6.3.3 extend()

```
a = [1, 2, 3]
b = [4, 5, 6]

a.extend(b)
print(a)  # [1, 2, 3, 4, 5, 6]
```

Equivalente a `a += b`.

6.3.4 remove()

```
frutas = ["manzana", "pera", "plátano", "pera"]
frutas.remove("pera")  # elimina la primera 'pera'
print(frutas)  # ['manzana', 'plátano', 'pera']
```

Si el elemento no existe, lanza **ValueError**.

6.3.5 pop()

```
numeros = [10, 20, 30, 40]

ultimo = numeros.pop()
print(ultimo)   # 40
print(numeros)  # [10, 20, 30]

primero = numeros.pop(0)
print(primero)  # 10
print(numeros)  # [20, 30]
```

6.3.6 clear()

```
numeros = [1, 2, 3]
numeros.clear()
print(numeros)  # []
```

6.3.7 index() y count()

```
letras = ["a", "b", "c", "b", "d"]

print(letras.index("b"))  # 1
print(letras.count("b"))  # 2
```

Si `index()` no encuentra el elemento, lanza **ValueError**.

6.3.8 reverse()

```
numeros = [1, 2, 3, 4]
numeros.reverse()
print(numeros)  # [4, 3, 2, 1]
```

6.4 Ordenación

Ordenar listas es una operación muy común. Python ofrece:

- El método **sort()** → ordena la lista **en el sitio** (modifica la lista).
- La función **sorted()** → devuelve una **nueva lista** ordenada, sin modificar la original.

6.4.1 sort() básico

```
numeros = [4, 1, 7, 3, 2]
numeros.sort()
```

```
print(numeros)  # [1, 2, 3, 4, 7]
```

6.4.2 sort(reverse=True)

```
numeros = [4, 1, 7, 3, 2]
numeros.sort(reverse=True)
print(numeros)  # [7, 4, 3, 2, 1]
```

6.4.3 sorted()

```
numeros = [4, 1, 7, 3, 2]
ordenada = sorted(numeros)

print("Original:", numeros)   # [4, 1, 7, 3, 2]
print("Ordenada:", ordenada)  # [1, 2, 3, 4, 7]
```

6.4.4 Ordenar cadenas

```
nombres = ["Luis", "ana", "Pedro", "carlos"]

nombres.sort()
print(nombres)  # orden lexicográfico, mayúsculas antes que minúsculas
```

Para ordenar sin distinguir mayúsculas/minúsculas:

```
nombres = ["Luis", "ana", "Pedro", "carlos"]

nombres.sort(key=str.lower)
print(nombres)  # ['ana', 'carlos', 'Luis', 'Pedro']
```

6.4.5 Ordenar por longitud

```
palabras = ["Python", "es", "genial", "y", "potente"]

palabras.sort(key=len)
print(palabras)  # ['y', 'es', 'Python', 'genial', 'potente']
```

6.5 Copias superficiales y profundas

Este es un tema **muy importante** y fuente de muchos errores.

Cuando hacemos:

```
a = [1, 2, 3]
```

```
b = a
```

No estamos creando una copia independiente.
`a` y `b` apuntan a **la misma lista** en memoria.

```
a = [1, 2, 3]
b = a


b[0] = 99
print(a)  # [99, 2, 3]
print(b)  # [99, 2, 3]
```

Para crear una **copia superficial** (shallow copy) de una lista, podemos usar:

- `lista.copy()`
- `lista[:]`
- `list(lista)`

6.5.1 Copia superficial

```
a = [1, 2, 3]
b = a.copy()


b[0] = 99
print(a)  # [1, 2, 3]
print(b)  # [99, 2, 3]
```

Esto funciona bien si la lista contiene **elementos simples** (números, cadenas, etc.).
Pero si contiene **listas dentro de listas**, la copia superficial **comparte** las sublistas.

```
a = [[1, 2], [3, 4]]
b = a.copy()


b[0][0] = 99
print(a)  # [[99, 2], [3, 4]]
print(b)  # [[99, 2], [3, 4]]
```

6.5.2 Copia profunda (deep copy)

Para copiar **toda la estructura**, incluyendo sublistas, usamos `copy.deepcopy()`:

```
import copy


a = [[1, 2], [3, 4]]
```

```
b = copy.deepcopy(a)


b[0][0] = 99
print(a)  # [[1, 2], [3, 4]]
print(b)  # [[99, 2], [3, 4]]
```

Resumen:

- **Asignación** (`b = a`) → misma lista.
- **Copia superficial** (`a.copy()`, `a[:]`) → nueva lista, pero sublistas compartidas.
- **Copia profunda** (`copy.deepcopy(a)`) → copia completa, sin compartir sublistas.

6.6 Listas anidadas

Una **lista anidada** es una lista que contiene otras listas como elementos.
Se usan para representar **tablas**, **matrices**, **tableros de juegos**, etc.

6.6.1 Ejemplo de lista anidada

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]


print(matriz[0])     # [1, 2, 3]
print(matriz[1][2])  # 6
```

6.6.2 Recorrer una lista anidada

```
for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print()
```

Salida:

```
1 2 3
4 5 6
7 8 9
```

6.6.3 Crear una matriz inicializada

Error típico:

```
filas = 3
columnas = 3
matriz = [[0] * columnas] * filas  # ¡Cuidado!
```

Esto crea **tres referencias a la misma sublista**.
Si modificas una fila, se modifican todas.

```
matriz[0][0] = 99
print(matriz)
# [[99, 0, 0], [99, 0, 0], [99, 0, 0]]
```

Forma correcta:

```
filas = 3
columnas = 3
matriz = [[0] * columnas for _ in range(filas)]

matriz[0][0] = 99
print(matriz)
# [[99, 0, 0], [0, 0, 0], [0, 0, 0]]
```

6.7 Errores comunes

Al trabajar con listas, hay errores muy frecuentes:

6.7.1 Índices fuera de rango

```
numeros = [1, 2, 3]
# print(numeros[10])  # IndexError
```

Solución: comprobar la longitud con `len()`.

6.7.2 Confundir copia con referencia

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a)  # [99, 2, 3]
```

Solución: usar `copy()` o slicing si quieres una copia.

6.7.3 Modificar la lista mientras se itera

```
numeros = [1, 2, 3, 4, 5]
```

```
for n in numeros:
    if n % 2 == 0:
        numeros.remove(n)
```

Puede producir resultados inesperados.
Solución: crear una nueva lista o iterar sobre una copia.

6.7.4 Usar `sort()` esperando una nueva lista

```
numeros = [3, 1, 2]
ordenada = numeros.sort()
print(ordenada)  # None
```

`sort()` **no devuelve** la lista ordenada, la modifica en el sitio.
Solución: usar `sorted()` si quieres una nueva lista.

6.7.5 Problemas con listas anidadas y copias

Ya visto: usar `[[0]*n]*m` crea filas compartidas.
Solución: usar list comprehensions para crear filas independientes.

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Crea una lista con los números del 1 al 5 y muestra su longitud.

Ejercicio 2
Crea una lista con los nombres de 4 personas y muestra el primer y el último nombre usando índices.

Ejercicio 3
Crea una lista con los números `[10, 20, 30, 40, 50]` y muestra los elementos del índice 1 al 3 usando slicing.

Ejercicio 4
Crea una lista con los números `[1, 2, 3, 4, 5]` y crea otra lista con los mismos elementos pero en orden inverso usando slicing.

Ejercicio 5
Pide al usuario 5 números enteros, guárdalos en una lista y muestra la suma de todos ellos.

Ejercicio 6
Dada la lista `[3, 1, 4, 1, 5, 9, 2]`, usa métodos de lista para:

- Añadir el número 6 al final.
- Insertar el número 0 al principio.
- Eliminar la primera aparición del número 1.

Ejercicio 7
Dada la lista `["manzana", "pera", "plátano"]`, añade `"naranja"` al final y `"kiwi"` en la posición 1.

Ejercicio 8
Dada la lista `[10, 20, 30, 40, 50]`, elimina el último elemento con `pop()` y muestra el valor eliminado y la lista resultante.

Ejercicio 9
Dada la lista `[1, 2, 3, 4, 5, 3, 3]`, muestra cuántas veces aparece el número 3 y el índice de su primera aparición.

Ejercicio 10
Dada la lista `[4, 2, 9, 1, 5]`, ordénala de menor a mayor y luego de mayor a menor.

Ejercicio 11
Crea una lista de palabras y ordénala por orden alfabético sin distinguir mayúsculas/minúsculas.

Ejercicio 12
Crea una lista de palabras y ordénala por longitud (de más corta a más larga).

Ejercicio 13
Crea una lista `a = [1, 2, 3]` y haz `b = a`. Modifica `b[0]` y observa qué pasa con `a`. Explica por qué.

Ejercicio 14
Repite el ejercicio anterior, pero esta vez crea `b` como una copia de `a` usando `copy()`. Modifica `b[0]` y observa qué pasa con `a`.

Ejercicio 15
Crea una lista anidada que represente una matriz 3×3 con los números del 1 al 9. Muestra el elemento de la fila 2, columna 3 (índices empezando en 0).

Ejercicio 16
Recorre la matriz del ejercicio anterior con bucles anidados y muestra todos sus elementos en forma de tabla.

Ejercicio 17
Crea una matriz 3×3 inicializada a 0 usando la forma incorrecta `[[0]*3]*3` y demuestra que al cambiar un elemento se cambian varias filas.

Ejercicio 18
Crea una matriz 3×3 inicializada a 0 usando una list comprehension correcta y demuestra que al cambiar un elemento solo cambia esa posición.

Ejercicio 19
Usa una list comprehension para crear una lista con los cuadrados de los números del 1 al 10.

Ejercicio 20
Usa una list comprehension para crear una lista con los números pares del 1 al 20 y otra con los impares del 1 al 20.

Soluciones de los 20 ejercicios

Solución 1

```
lista = [1, 2, 3, 4, 5]
print("Lista:", lista)
print("Longitud:", len(lista))
```

Solución 2

```
nombres = ["Ana", "Luis", "María", "Pedro"]

primero = nombres[0]
ultimo = nombres[-1]

print("Primer nombre:", primero)
print("Último nombre:", ultimo)
```

Solución 3

```
numeros = [10, 20, 30, 40, 50]

sublista = numeros[1:4]  # índices 1, 2, 3
print("Sublista:", sublista)
```

Solución 4

```
original = [1, 2, 3, 4, 5]
invertida = original[::-1]

print("Original:", original)
print("Invertida:", invertida)
```

Solución 5

```
numeros = []

for i in range(5):
```

```
    n = int(input(f"Introduce el número {i+1}: "))
    numeros.append(n)

suma = sum(numeros)

print("Lista:", numeros)
print("Suma:", suma)
```

Solución 6

```
lista = [3, 1, 4, 1, 5, 9, 2]

lista.append(6)
lista.insert(0, 0)
lista.remove(1)  # elimina la primera aparición de 1

print("Lista resultante:", lista)
```

Solución 7

```
frutas = ["manzana", "pera", "plátano"]

frutas.append("naranja")
frutas.insert(1, "kiwi")

print("Lista de frutas:", frutas)
```

Solución 8

```
numeros = [10, 20, 30, 40, 50]

eliminado = numeros.pop()
print("Elemento eliminado:", eliminado)
print("Lista resultante:", numeros)
```

Solución 9

```
lista = [1, 2, 3, 4, 5, 3, 3]

veces = lista.count(3)
indice = lista.index(3)

print("El número 3 aparece", veces, "veces.")
print("La primera aparición está en el índice:", indice)
```

Solución 10

```
numeros = [4, 2, 9, 1, 5]

numeros.sort()
print("Orden ascendente:", numeros)

numeros.sort(reverse=True)
print("Orden descendente:", numeros)
```

Solución 11

```
palabras = ["Python", "java", "C", "ruby", "Go"]

palabras.sort(key=str.lower)
print("Palabras ordenadas (sin distinguir mayúsculas):", palabras)
```

Solución 12

```
palabras = ["Python", "es", "genial", "y", "potente"]

palabras.sort(key=len)
print("Palabras ordenadas por longitud:", palabras)
```

Solución 13

```
a = [1, 2, 3]
```

```
b = a  # b referencia la misma lista

b[0] = 99

print("a:", a)
print("b:", b)
```

Explicación: a y b apuntan a la **misma lista**. Cambiar b también cambia a.

Solución 14

```
a = [1, 2, 3]
b = a.copy()  # copia superficial

b[0] = 99

print("a:", a)
print("b:", b)
```

Explicación: ahora a y b son listas distintas. Cambiar b no afecta a a.

Solución 15

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# fila 2, columna 3 (índices empezando en 0) => matriz[1][2]
elemento = matriz[1][2]
print("Elemento fila 2, columna 3:", elemento)
```

Solución 16

```
matriz = [
    [1, 2, 3],
```

```
    [4, 5, 6],
    [7, 8, 9]
]

for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print()
```

Solución 17

```
filas = 3
columnas = 3

matriz = [[0] * columnas] * filas  # forma incorrecta

matriz[0][0] = 99

print("Matriz (forma incorrecta):")
for fila in matriz:
    print(fila)
```

Verás algo como:

```
[99, 0, 0]
[99, 0, 0]
[99, 0, 0]
```

Todas las filas comparten la misma sublista.

Solución 18

```
filas = 3
columnas = 3

matriz = [[0] * columnas for _ in range(filas)]  # forma correcta
```

```
matriz[0][0] = 99

print("Matriz (forma correcta):")
for fila in matriz:
    print(fila)
```

Salida:

```
[99, 0, 0]
[0, 0, 0]
[0, 0, 0]
```

Cada fila es independiente.

Solución 19

```
cuadrados = [x**2 for x in range(1, 11)]
print("Cuadrados del 1 al 10:", cuadrados)
```

Solución 20

```
pares = [x for x in range(1, 21) if x % 2 == 0]
impares = [x for x in range(1, 21) if x % 2 != 0]

print("Pares:", pares)
print("Impares:", impares)
```

---

7. Conjuntos (sets) en Python

Los **conjuntos** (`set`) en Python son estructuras de datos muy potentes cuando necesitamos trabajar con **colecciones de elementos sin duplicados**, hacer **operaciones de teoría de conjuntos** (unión, intersección, diferencia…), comprobar **pertenencia** de forma rápida, o eliminar duplicados de listas.

En esta lección veremos, con teoría muy detallada y muchos ejemplos:

- **7.1 Creación de sets**
- **7.2 Operaciones matemáticas**
- **7.3 Métodos principales**
- **7.4 Conjuntos inmutables (frozenset)**
- **7.5 Usos prácticos**
- **7.6 Errores típicos**

Al final tendrás **20 ejercicios** con su **solución detallada**.

7.1 Creación de sets

Un **set** es una colección **no ordenada**, **sin elementos duplicados** y **mutable** (se puede modificar). Se define usando llaves `{}` o la función `set()`.

7.1.1 Creación básica

```
vacio = set()                 # conjunto vacío
numeros = {1, 2, 3, 4, 5}     # conjunto de enteros
letras = {"a", "b", "c"}      # conjunto de cadenas

print(vacio)
print(numeros)
print(letras)
```

Importante: `{}` sin nada dentro crea un **diccionario**, no un set.
Por eso, el conjunto vacío se crea con `set()`.

```
vacio = {}
print(type(vacio))  # dict

vacio_set = set()
print(type(vacio_set))  # set
```

7.1.2 Eliminación automática de duplicados

Si creas un set con elementos repetidos, los duplicados se eliminan automáticamente:

```
numeros = {1, 2, 2, 3, 3, 3, 4}
```

```
print(numeros)  # {1, 2, 3, 4}
```

El orden de impresión puede variar, porque los sets son **no ordenados**.

7.1.3 Crear un set a partir de otras colecciones

Podemos convertir listas, tuplas, cadenas, etc., en sets usando `set()`:

```
lista = [1, 2, 2, 3, 3, 4]
conjunto = set(lista)
print(conjunto)  # {1, 2, 3, 4}

texto = "abracadabra"
letras = set(texto)
print(letras)  # {'a', 'b', 'r', 'c', 'd'} (orden no garantizado)
```

7.2 Operaciones matemáticas

Los sets implementan las operaciones clásicas de **teoría de conjuntos**:

- **Unión**
- **Intersección**
- **Diferencia**
- **Diferencia simétrica**

7.2.1 Unión

La **unión** de dos conjuntos A y B es el conjunto de elementos que están en A, en B o en ambos.

En Python:

- Operador `|`
- Método `.union()`

```
A = {1, 2, 3}
B = {3, 4, 5}

union1 = A | B
union2 = A.union(B)

print(union1)  # {1, 2, 3, 4, 5}
print(union2)  # {1, 2, 3, 4, 5}
```

7.2.2 Intersección

La **intersección** de A y B es el conjunto de elementos que están en **ambos**.

- Operador &
- Método `.intersection()`

```
A = {1, 2, 3}
B = {3, 4, 5}

inter1 = A & B
inter2 = A.intersection(B)

print(inter1)  # {3}
print(inter2)  # {3}
```

7.2.3 Diferencia

La **diferencia** A − B es el conjunto de elementos que están en A pero **no** en B.

- Operador `-`
- Método `.difference()`

```
A = {1, 2, 3, 4}
B = {3, 4, 5}

dif1 = A - B
dif2 = A.difference(B)

print(dif1)  # {1, 2}
print(dif2)  # {1, 2}
```

7.2.4 Diferencia simétrica

La **diferencia simétrica** de A y B es el conjunto de elementos que están en A o en B, pero **no en ambos**.

- Operador `^`
- Método `.symmetric_difference()`

```
A = {1, 2, 3}
B = {3, 4, 5}

dif_sim1 = A ^ B
dif_sim2 = A.symmetric_difference(B)
```

```
print(dif_sim1)  # {1, 2, 4, 5}
print(dif_sim2)  # {1, 2, 4, 5}
```

7.2.5 Subconjuntos y superconjuntos

- `A.issubset(B)` → A es subconjunto de B
- `A.issuperset(B)` → A es superconjunto de B

```
A = {1, 2}
B = {1, 2, 3, 4}

print(A.issubset(B))   # True
print(B.issuperset(A)) # True
```

También existen los operadores <=, <, >=, >:

```
print(A <= B)  # True (subconjunto)
print(A < B)   # True (subconjunto propio)
```

7.3 Métodos principales

Los sets tienen métodos para **añadir**, **eliminar**, **actualizar** y **consultar** elementos.

Los más importantes:

- **add(x)** → añade un elemento.
- **update(iterable)** → añade varios elementos.
- **remove(x)** → elimina un elemento (error si no existe).
- **discard(x)** → elimina un elemento (sin error si no existe).
- **pop()** → elimina y devuelve un elemento arbitrario.
- **clear()** → vacía el conjunto.
- **copy()** → copia superficial.

7.3.1 add()

```
s = {1, 2, 3}
s.add(4)
s.add(2)  # no pasa nada, ya está

print(s)  # {1, 2, 3, 4}
```

7.3.2 update()

```
s = {1, 2, 3}
s.update([3, 4, 5])
print(s)  # {1, 2, 3, 4, 5}
```

`update()` acepta cualquier iterable: listas, tuplas, otros sets…

7.3.3 remove() y discard()

```
s = {1, 2, 3}

s.remove(2)
print(s)  # {1, 3}

# s.remove(4)  # KeyError: 4
```

`discard()` no lanza error si el elemento no existe:

```
s = {1, 2, 3}

s.discard(2)
s.discard(4)  # no pasa nada

print(s)  # {1, 3}
```

7.3.4 pop()

`pop()` elimina y devuelve un elemento **arbitrario** (no hay orden):

```
s = {10, 20, 30}

elem = s.pop()
print("Elemento extraído:", elem)
print("Conjunto restante:", s)
```

7.3.5 clear() y copy()

```
s = {1, 2, 3}
copia = s.copy()

s.clear()

print("Original:", s)  # set()
print("Copia:", copia) # {1, 2, 3}
```

7.4 Conjuntos inmutables (frozenset)

Además de `set`, Python tiene **conjuntos inmutables** llamados **`frozenset`**.

- Un `frozenset` **no se puede modificar** después de crearlo.
- Se puede usar como **clave de diccionario** o como elemento de otro set (porque es hashable).

7.4.1 Creación de frozenset

```
s = {1, 2, 3}
f = frozenset(s)

print(f)          # frozenset({1, 2, 3})
print(type(f))    # <class 'frozenset'>
```

También se puede crear directamente:

```
f = frozenset([1, 2, 3, 3])
print(f)  # frozenset({1, 2, 3})
```

7.4.2 Operaciones con frozenset

Los `frozenset` soportan las mismas **operaciones matemáticas** que los sets:

```
A = frozenset({1, 2, 3})
B = frozenset({3, 4, 5})

print(A | B)  # unión
print(A & B)  # intersección
print(A - B)  # diferencia
```

Pero **no** tienen métodos que modifiquen el conjunto (`add`, `remove`, etc.), porque son inmutables.

7.5 Usos prácticos

Los sets son muy útiles en muchos contextos reales.

7.5.1 Eliminar duplicados de una lista

```
lista = [1, 2, 2, 3, 3, 3, 4]
sin_duplicados = list(set(lista))

print(sin_duplicados)  # [1, 2, 3, 4] (orden no garantizado)
```

Si quieres mantener el **orden de aparición**, puedes hacer:

```
lista = [1, 2, 2, 3, 3, 3, 4]
```

```
vistos = set()
resultado = []

for x in lista:
    if x not in vistos:
        vistos.add(x)
        resultado.append(x)

print(resultado)  # [1, 2, 3, 4]
```

7.5.2 Comprobar pertenencia rápida

La operación `x in conjunto` es muy rápida (búsqueda por hash).

```
usuarios_registrados = {"ana", "luis", "maria"}

nombre = input("Introduce tu nombre: ")

if nombre in usuarios_registrados:
    print("Usuario registrado.")
else:
    print("Usuario no registrado.")
```

7.5.3 Comparar conjuntos de datos

Por ejemplo, alumnos que han entregado una tarea, alumnos matriculados, etc.

```
matriculados = {"Ana", "Luis", "María", "Pedro"}
entregaron = {"Ana", "Pedro"}

no_entregaron = matriculados - entregaron
print("No entregaron:", no_entregaron)
```

7.5.4 Detectar elementos comunes

```
grupo1 = {"rock", "jazz", "blues"}
grupo2 = {"pop", "jazz", "electrónica"}

comunes = grupo1 & grupo2
print("Géneros en común:", comunes)
```

7.6 Errores típicos

Trabajando con sets es fácil cometer ciertos errores. Veamos los más frecuentes.

7.6.1 Usar {} pensando que es un set vacío

```
vacio = {}
print(type(vacio))  # dict
```

Solución: usar `set()` para crear un conjunto vacío.

7.6.2 Intentar indexar un set

Los sets son **no ordenados** y **no indexables**:

```
s = {1, 2, 3}
# print(s[0])  # TypeError
```

Si necesitas acceder por índice, convierte a lista:

```
lista = list(s)
print(lista[0])
```

7.6.3 Esperar un orden concreto

```
s = {1, 2, 3, 4, 5}
print(s)  # el orden puede variar
```

No debes depender del orden de un set. Si necesitas orden, usa una lista ordenada.

7.6.4 Modificar un set mientras se itera

```
s = {1, 2, 3, 4, 5}

for x in s:
    if x % 2 == 0:
        s.remove(x)  # puede dar problemas
```

Modificar un set mientras se recorre puede producir errores o comportamientos inesperados.
Solución: crear un nuevo set o iterar sobre una copia:

```
s = {1, 2, 3, 4, 5}
a_eliminar = {x for x in s if x % 2 == 0}

for x in a_eliminar:
    s.remove(x)
```

```
print(s)  # {1, 3, 5}
```

7.6.5 Intentar meter elementos no hashables

Los elementos de un set deben ser **hashables** (inmutables): números, cadenas, tuplas, frozenset…
No se pueden meter listas ni otros sets mutables:

```
# s = {[1, 2], [3, 4]}  # TypeError: unhashable type: 'list'
```

Si necesitas conjuntos de conjuntos, usa `frozenset`:

```
s = {frozenset({1, 2}), frozenset({3, 4})}
print(s)
```

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Crea un conjunto con los números `{1, 2, 3, 4, 5}` y muestra su tipo y su contenido.

Ejercicio 2
Crea un conjunto a partir de la lista `[1, 2, 2, 3, 3, 4, 4, 5]` y muestra el conjunto resultante.

Ejercicio 3
Pide al usuario una palabra y crea un conjunto con las letras que aparecen en ella. Muestra el conjunto.

Ejercicio 4
Dados los conjuntos `A = {1, 2, 3}` y `B = {3, 4, 5}`, calcula y muestra:

- La unión
- La intersección
- La diferencia A − B
- La diferencia B − A
- La diferencia simétrica

Ejercicio 5
Dados los conjuntos `A = {1, 2}` y `B = {1, 2, 3, 4}`, comprueba si A es subconjunto de B y si B es superconjunto de A.

Ejercicio 6
Crea un conjunto vacío y añade los números del 1 al 5 usando `add()` en un bucle.

Ejercicio 7
Crea un conjunto con los números `{1, 2, 3, 4, 5}` y elimina el número 3 usando `remove()`. Luego intenta eliminar el número 10 usando `discard()` y observa la diferencia.

Ejercicio 8
Crea un conjunto con los números `{10, 20, 30, 40}` y usa `pop()` para extraer un elemento. Muestra el elemento extraído y el conjunto resultante.

Ejercicio 9
Crea un conjunto con los nombres `{"Ana", "Luis", "María"}` y pide al usuario un nombre. Muestra si el nombre está en el conjunto o no.

Ejercicio 10
Dada la lista `[1, 2, 2, 3, 3, 3, 4, 4, 5]`, elimina los duplicados usando un set y muestra la lista resultante sin duplicados.

Ejercicio 11
Dadas dos listas de nombres, crea un set con cada una y muestra los nombres que aparecen en ambas listas (intersección).

Ejercicio 12
Dadas dos listas de nombres, muestra los nombres que aparecen en la primera lista pero no en la segunda (diferencia).

Ejercicio 13
Crea un `frozenset` a partir del conjunto `{1, 2, 3}` y muestra su tipo. Intenta usar `add()` sobre el `frozenset` y observa qué ocurre.

Ejercicio 14
Crea un conjunto de géneros musicales favoritos de una persona y otro conjunto de géneros de otra persona. Muestra los géneros en común y los géneros que solo le gustan a una de las dos (diferencia simétrica).

Ejercicio 15
Pide al usuario una frase y muestra cuántas letras distintas contiene (usa un set de caracteres).

Ejercicio 16
Pide al usuario una lista de números separados por comas, conviértela en lista de enteros y luego en set. Muestra cuántos números distintos ha introducido.

Ejercicio 17
Crea un conjunto con los números del 1 al 10 y otro con los números pares del 1 al 10. Muestra el conjunto de números impares usando operaciones de conjuntos.

Ejercicio 18
Crea un conjunto con algunos números y recórrelo con un `for` para mostrar sus elementos. Explica por qué el orden puede cambiar entre ejecuciones.

Ejercicio 19
Crea un conjunto de palabras y demuestra que no se pueden añadir listas como elementos, pero sí tuplas.

Ejercicio 20
Crea un conjunto de alumnos matriculados y otro de alumnos que han entregado un trabajo. Muestra:

- Los alumnos que han entregado.
- Los que no han entregado.
- Los que han entregado pero no estaban matriculados (por error).

Soluciones de los 20 ejercicios

Solución 1

```
s = {1, 2, 3, 4, 5}
print("Tipo:", type(s))
print("Contenido:", s)
```

Solución 2

```
lista = [1, 2, 2, 3, 3, 4, 4, 5]
conjunto = set(lista)

print("Conjunto sin duplicados:", conjunto)
```

Solución 3

```
palabra = input("Introduce una palabra: ")

letras = set(palabra)
print("Letras distintas:", letras)
```

Solución 4

```
A = {1, 2, 3}
B = {3, 4, 5}

union = A | B
inter = A & B
dif_A_B = A - B
dif_B_A = B - A
dif_sim = A ^ B

print("Unión:", union)
print("Intersección:", inter)
print("Diferencia A - B:", dif_A_B)
print("Diferencia B - A:", dif_B_A)
```

```
print("Diferencia simétrica:", dif_sim)
```

Solución 5

```
A = {1, 2}
B = {1, 2, 3, 4}

print("A es subconjunto de B:", A.issubset(B))
print("B es superconjunto de A:", B.issuperset(A))
```

Solución 6

```
s = set()

for i in range(1, 6):
    s.add(i)

print("Conjunto:", s)
```

Solución 7

```
s = {1, 2, 3, 4, 5}

s.remove(3)
print("Después de remove(3):", s)

s.discard(10)  # no lanza error
print("Después de discard(10):", s)
```

Solución 8

```
s = {10, 20, 30, 40}

elem = s.pop()
print("Elemento extraído:", elem)
print("Conjunto restante:", s)
```

Solución 9

```
usuarios = {"Ana", "Luis", "María"}

nombre = input("Introduce un nombre: ")

if nombre in usuarios:
    print("El nombre está en el conjunto.")
else:
    print("El nombre NO está en el conjunto.")
```

Solución 10

```
lista = [1, 2, 2, 3, 3, 3, 4, 4, 5]

sin_duplicados = list(set(lista))
print("Lista sin duplicados:", sin_duplicados)
```

Solución 11

```
lista1 = ["Ana", "Luis", "María", "Pedro"]
lista2 = ["Luis", "Carlos", "María", "Elena"]

set1 = set(lista1)
set2 = set(lista2)

comunes = set1 & set2

print("Nombres en ambas listas:", comunes)
```

Solución 12

```
lista1 = ["Ana", "Luis", "María", "Pedro"]
lista2 = ["Luis", "Carlos", "María", "Elena"]
```

```
set1 = set(lista1)
set2 = set(lista2)

solo_en_1 = set1 - set2

print("Nombres solo en la primera lista:", solo_en_1)
```

Solución 13

```
s = {1, 2, 3}
f = frozenset(s)

print("frozenset:", f)
print("Tipo:", type(f))

# f.add(4)  # AttributeError: 'frozenset' object has no attribute
'add'
```

No se puede modificar un `frozenset`.

Solución 14

```
persona1 = {"rock", "jazz", "blues"}
persona2 = {"pop", "jazz", "electrónica", "rock"}

comunes = persona1 & persona2
dif_sim = persona1 ^ persona2

print("Géneros en común:", comunes)
print("Géneros que solo le gustan a una de las dos personas:",
dif_sim)
```

Solución 15

```
frase = input("Introduce una frase: ")
```

```
caracteres = set(frase.replace(" ", ""))  # opcional: quitar
espacios
print("Número de letras/caracteres distintos:", len(caracteres))
print("Conjunto de caracteres:", caracteres)
```

Solución 16

```
cadena = input("Introduce números separados por comas: ")

# Ejemplo: "1,2,2,3,4,4"
partes = cadena.split(",")
numeros = [int(p.strip()) for p in partes]

conjunto = set(numeros)

print("Has introducido", len(numeros), "números.")
print("Hay", len(conjunto), "números distintos.")
print("Conjunto de números distintos:", conjunto)
```

Solución 17

```
todos = set(range(1, 11))
pares = {2, 4, 6, 8, 10}

impares = todos - pares

print("Todos:", todos)
print("Pares:", pares)
print("Impares:", impares)
```

Solución 18

```
s = {10, 20, 30, 40, 50}

for x in s:
```

```
    print(x)
```

Explicación: los sets son colecciones **no ordenadas**, por lo que el orden de los elementos al recorrerlos puede variar entre ejecuciones.

Solución 19

```
s = set()

# s.add([1, 2])  # TypeError: unhashable type: 'list'

s.add((1, 2))  # tupla, sí es hashable
s.add((3, 4))

print("Conjunto con tuplas:", s)
```

No se pueden añadir listas (mutables), pero sí tuplas (inmutables).

Solución 20

```
matriculados = {"Ana", "Luis", "María", "Pedro"}
entregaron = {"Ana", "Pedro", "Carlos"}

han_entregado = matriculados & entregaron
no_entregaron = matriculados - entregaron
entregaron_no_matriculados = entregaron - matriculados

print("Han entregado (y están matriculados):", han_entregado)
print("No han entregado:", no_entregaron)
print("Entregaron pero no estaban matriculados (error):",
entregaron_no_matriculados)
```

---

8. Tuplas en Python

Las **tuplas** son una estructura de datos fundamental en Python, muy parecida a las listas, pero con una diferencia clave: **son inmutables**.
Eso significa que, una vez creada una tupla, **no se puede modificar** (no se pueden añadir, eliminar ni cambiar elementos).

Esta característica las hace ideales para representar **colecciones de datos fijos**, coordenadas, fechas, claves de diccionarios, resultados múltiples de funciones, etc.

En esta lección veremos:

- **8.1 Creación y propiedades**
- **8.2 Indexación y slicing**
- **8.3 Desempaquetado**
- **8.4 Tuplas anidadas**
- **8.5 Usos prácticos**
- **8.6 Errores frecuentes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

8.1 Creación y propiedades

Una **tupla** es una colección **ordenada**, **inmutable** y que puede contener **elementos de cualquier tipo** (números, cadenas, listas, otras tuplas, etc.).

Se define usando **paréntesis** `()` y separando los elementos con comas.

8.1.1 Creación básica de tuplas

```
vacia = ()                          # tupla vacía
numeros = (1, 2, 3, 4, 5)           # tupla de enteros
nombres = ("Ana", "Luis", "Eva")    # tupla de cadenas
mixta = (1, "hola", 3.14, True)     # tupla con tipos mezclados

print(vacia)
print(numeros)
print(nombres)
print(mixta)
```

8.1.2 Tupla de un solo elemento

Aquí hay un detalle importante: para crear una tupla de **un solo elemento**, hay que poner una **coma**.

```
t1 = (5)
t2 = (5,)
```

```
print(type(t1))  # int
print(type(t2))  # tuple
```

- **t1** es un entero, no una tupla.
- **t2** es una tupla con un solo elemento.

También se puede escribir sin paréntesis:

```
t3 = 5,
print(type(t3))  # tuple
```

8.1.3 Tuplas sin paréntesis (packing)

Python permite crear tuplas **sin paréntesis**, solo separando por comas:

```
t = 1, 2, 3
print(t)         # (1, 2, 3)
print(type(t))   # tuple
```

Esto se llama **empaquetado** (packing).

8.1.4 Longitud de una tupla

Como con las listas, usamos `len()`:

```
numeros = (10, 20, 30)
print(len(numeros))  # 3
```

8.1.5 Inmutabilidad

La propiedad clave: **no se puede modificar** una tupla.

```
t = (1, 2, 3)
# t[0] = 99  # TypeError: 'tuple' object does not support item assignment
```

No se pueden usar métodos como `append`, `remove`, etc.
Si necesitas una colección modificable, usa una **lista**.

8.2 Indexación y slicing

Las tuplas son **secuencias**, igual que las listas y las cadenas.
Por tanto, soportan **índices** y **slicing**.

8.2.1 Indexación

Los índices empiezan en **0**:

```
letras = ("a", "b", "c", "d", "e")


print(letras[0])   # 'a'
```

```
print(letras[1])   # 'b'
print(letras[4])   # 'e'
```

Índices negativos:

```
print(letras[-1])  # 'e'
print(letras[-2])  # 'd'
print(letras[-5])  # 'a'
```

Si accedemos a un índice fuera de rango:

```
# print(letras[10])  # IndexError
```

8.2.2 Slicing (rebanado)

La sintaxis es igual que en listas:

```
subtupla = letras[inicio:fin:paso]
```

Ejemplos:

```
letras = ("a", "b", "c", "d", "e", "f")

print(letras[0:3])   # ('a', 'b', 'c')
print(letras[2:5])   # ('c', 'd', 'e')
print(letras[:4])    # ('a', 'b', 'c', 'd')
print(letras[3:])    # ('d', 'e', 'f')
print(letras[:])     # copia completa
```

Con paso:

```
print(letras[0:6:2])  # ('a', 'c', 'e')
print(letras[::2])    # ('a', 'c', 'e')
print(letras[::-1])   # ('f', 'e', 'd', 'c', 'b', 'a')
```

Recuerda: el slicing devuelve **otra tupla**.

8.3 Desempaquetado

El **desempaquetado** consiste en asignar los elementos de una tupla a varias variables de una sola vez.

8.3.1 Desempaquetado básico

```
punto = (3, 4)

x, y = punto
```

```
print("x =", x)  # 3
print("y =", y)  # 4
```

El número de variables debe coincidir con el número de elementos.

```
t = (1, 2, 3)
a, b, c = t
print(a, b, c)  # 1 2 3
```

8.3.2 Intercambio de variables

Una aplicación muy típica:

```
a = 10
b = 20

a, b = b, a

print(a, b)  # 20 10
```

Python crea una tupla `(b, a)` y luego la desempaqueta en `a` y `b`.

8.3.3 Desempaquetado con *

Podemos usar el operador `*` para capturar “el resto” de elementos.

```
t = (1, 2, 3, 4, 5)

primero, *medio, ultimo = t

print(primero)  # 1
print(medio)    # [2, 3, 4]
print(ultimo)   # 5
```

Observa que `medio` es una **lista**, no una tupla.

Otro ejemplo:

```
a, b, *resto = (10, 20, 30, 40, 50)
print(a)      # 10
print(b)      # 20
print(resto)  # [30, 40, 50]
```

8.4 Tuplas anidadas

Una tupla puede contener otras tuplas (o listas, o cualquier objeto).
Esto permite representar estructuras más complejas: matrices, puntos 3D, etc.

8.4.1 Ejemplo de tuplas anidadas

```
puntos = (
    (1, 2),
    (3, 4),
    (5, 6)
)

print(puntos[0])     # (1, 2)
print(puntos[1][0])  # 3
```

8.4.2 Recorrer tuplas anidadas

```
puntos = (
    (1, 2),
    (3, 4),
    (5, 6)
)

for x, y in puntos:
    print(f"Punto: x={x}, y={y}")
```

Salida:

```
Punto: x=1, y=2
Punto: x=3, y=4
Punto: x=5, y=6
```

8.4.3 Tuplas dentro de listas y listas dentro de tuplas

Es muy común mezclar estructuras:

```
lista_de_tuplas = [(1, 2), (3, 4), (5, 6)]
tupla_de_listas = ([1, 2], [3, 4], [5, 6])

print(lista_de_tuplas)
print(tupla_de_listas)
```

- En `lista_de_tuplas`, la lista es mutable, pero las tuplas internas no.
- En `tupla_de_listas`, la tupla es inmutable, pero las listas internas sí se pueden modificar:

```
tupla_de_listas[0][0] = 99
print(tupla_de_listas)  # ([99, 2], [3, 4], [5, 6])
```

8.5 Usos prácticos

Aunque las listas son más usadas para datos cambiantes, las tuplas tienen **usos muy importantes**.

8.5.1 Datos que no deben cambiar

Si tienes una colección de datos que **no debería modificarse**, usar una tupla es una forma de expresar esa intención.

```
DIAS_SEMANA = ("lunes", "martes", "miércoles", "jueves",
"viernes", "sábado", "domingo")
```

8.5.2 Coordenadas, puntos, tamaños

```
punto = (10, 20)
resolucion = (1920, 1080)
color_rgb = (255, 128, 0)
```

Son datos que conceptualmente tienen un número fijo de componentes.

8.5.3 Tuplas como claves de diccionarios

Las tuplas son **hashables** (si sus elementos también lo son), por lo que se pueden usar como **claves** en diccionarios.
Esto es muy útil para representar, por ejemplo, coordenadas como clave.

```
poblacion_ciudades = {
    ("España", "Zaragoza"): 675301,
    ("España", "Madrid"): 3223334,
    ("Francia", "París"): 2148327
}

print(poblacion_ciudades[("España", "Zaragoza")])
```

8.5.4 Devolver múltiples valores desde una función

Una función puede devolver una tupla con varios valores:

```
def dividir(dividendo, divisor):
    cociente = dividendo // divisor
    resto = dividendo % divisor
```

```
    return cociente, resto

q, r = dividir(17, 5)
print("Cociente:", q)
print("Resto:", r)
```

Internamente, `return cociente, resto` devuelve una tupla.

8.5.5 Iterar con índice y valor: enumerate()

`enumerate()` devuelve tuplas `(indice, valor)`:

```
frutas = ["manzana", "pera", "plátano"]

for indice, fruta in enumerate(frutas):
    print(indice, "=>", fruta)
```

8.6 Errores frecuentes

Veamos los errores más típicos al trabajar con tuplas.

8.6.1 Confundir tupla de un elemento con un valor simple

```
t1 = (5)
t2 = (5,)

print(type(t1))  # int
print(type(t2))  # tuple
```

Solución: recuerda poner la **coma** para tuplas de un solo elemento.

8.6.2 Intentar modificar una tupla

```
t = (1, 2, 3)
# t[0] = 99  # TypeError
```

Si necesitas modificar, convierte a lista:

```
t = (1, 2, 3)
lista = list(t)
lista[0] = 99
t_modificada = tuple(lista)
print(t_modificada)  # (99, 2, 3)
```

8.6.3 Olvidar el número correcto de variables al desempaquetar

```
t = (1, 2, 3)
# a, b = t      # ValueError: too many values to unpack
```

Solución: usar el mismo número de variables, o el operador `*`:

```
a, b, c = t
# o
a, *resto = t
```

8.6.4 Confundir lista y tupla en funciones

A veces se espera una lista y se pasa una tupla, o al revés.
En muchos casos no hay problema, pero si se usan métodos de lista (`append`, `remove`, etc.), fallará con tuplas.

```
t = (1, 2, 3)
# t.append(4)  # AttributeError
```

8.6.5 Pensar que la inmutabilidad es "seguridad total"

Si una tupla contiene **objetos mutables** (por ejemplo, listas), esos objetos sí se pueden modificar:

```
t = ([1, 2], [3, 4])
t[0][0] = 99
print(t)  # ([99, 2], [3, 4])
```

La tupla sigue siendo inmutable (no puedes cambiar sus referencias), pero los objetos internos pueden cambiar.

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Crea una tupla con los números del 1 al 5 y muestra su longitud.

Ejercicio 2
Crea una tupla con los nombres de 4 personas y muestra el primer y el último nombre usando índices.

Ejercicio 3
Crea una tupla con los números `(10, 20, 30, 40, 50)` y muestra los elementos del índice 1 al 3 usando slicing.

Ejercicio 4
Crea una tupla con los números `(1, 2, 3, 4, 5)` y crea otra tupla con los mismos elementos pero en orden inverso usando slicing.

Ejercicio 5
Crea una tupla con los días de la semana y muestra el tercer día y el último día.

Ejercicio 6
Pide al usuario 3 números enteros, guárdalos en una tupla y muestra la suma de todos ellos.

Ejercicio 7
Dada la tupla `(3, 1, 4, 1, 5, 9, 2)`, muestra cuántas veces aparece el número 1 (usando el método `count` de las tuplas).

Ejercicio 8
Dada la tupla `("manzana", "pera", "plátano", "pera")`, muestra el índice de la primera aparición de `"pera"` (usando `index`).

Ejercicio 9
Crea una tupla de un solo elemento con el número 5 y muestra su tipo para comprobar que es una tupla.

Ejercicio 10
Crea una tupla con los números `(1, 2, 3)` y conviértela en lista. Modifica el segundo elemento de la lista y vuelve a convertirla en tupla. Muestra el resultado.

Ejercicio 11
Crea una tupla `punto = (4, 7)` y desempaquétala en las variables `x` e `y`. Muestra los valores de `x` e `y`.

Ejercicio 12
Dada la tupla `(1, 2, 3, 4, 5)`, usa desempaquetado con `*` para guardar el primer elemento en una variable, el último en otra y el resto en una lista.

Ejercicio 13
Crea una tupla anidada que represente una matriz 2×2: `((1, 2), (3, 4))`. Muestra el elemento de la fila 2, columna 1 (índices empezando en 0).

Ejercicio 14
Recorre la tupla anidada del ejercicio anterior con un bucle `for` y muestra todos sus elementos.

Ejercicio 15
Crea un diccionario que use como clave una tupla `(pais, ciudad)` y como valor la población. Añade al menos 3 entradas y muestra la población de una ciudad concreta.

Ejercicio 16
Escribe una función que reciba dos números y devuelva una tupla con el resultado de la suma y el producto. Llama a la función y desempaqueta el resultado en dos variables.

Ejercicio 17
Dada una lista de tuplas `[(1, 2), (3, 4), (5, 6)]`, recórrela con un `for` y muestra la suma de cada par.

Ejercicio 18
Crea una tupla con algunos números y demuestra que no puedes usar `append` ni `remove` sobre ella (muestra el error comentado en el código).

Ejercicio 19
Crea una tupla que contenga dos listas: `([1, 2], [3, 4])`. Modifica el primer elemento de la primera lista y muestra la tupla resultante. Explica qué ha ocurrido.

Ejercicio 20
Pide al usuario una frase y crea una tupla con las palabras (usando `split`). Muestra cuántas palabras tiene la frase y cuál es la primera y la última.

Soluciones de los 20 ejercicios

Solución 1

```
t = (1, 2, 3, 4, 5)
print("Tupla:", t)
print("Longitud:", len(t))
```

Solución 2

```
nombres = ("Ana", "Luis", "María", "Pedro")

primero = nombres[0]
ultimo = nombres[-1]

print("Primer nombre:", primero)
print("Último nombre:", ultimo)
```

Solución 3

```
t = (10, 20, 30, 40, 50)

subtupla = t[1:4]  # índices 1, 2, 3
print("Subtupla:", subtupla)
```

Solución 4

```
t = (1, 2, 3, 4, 5)
invertida = t[::-1]

print("Original:", t)
```

```
print("Invertida:", invertida)
```

Solución 5

```
dias = ("lunes", "martes", "miércoles", "jueves", "viernes",
"sábado", "domingo")


tercer_dia = dias[2]
ultimo_dia = dias[-1]


print("Tercer día:", tercer_dia)
print("Último día:", ultimo_dia)
```

Solución 6

```
n1 = int(input("Introduce el primer número: "))
n2 = int(input("Introduce el segundo número: "))
n3 = int(input("Introduce el tercer número: "))


t = (n1, n2, n3)
suma = n1 + n2 + n3


print("Tupla:", t)
print("Suma:", suma)
```

Solución 7

```
t = (3, 1, 4, 1, 5, 9, 2)


veces = t.count(1)
print("El número 1 aparece", veces, "veces.")
```

Solución 8

```
t = ("manzana", "pera", "plátano", "pera")
```

```
indice = t.index("pera")
print("La primera 'pera' está en el índice:", indice)
```

Solución 9

```
t = (5,)
print("Tupla:", t)
print("Tipo:", type(t))
```

Solución 10

```
t = (1, 2, 3)
lista = list(t)

lista[1] = 99

t_modificada = tuple(lista)

print("Tupla original:", t)
print("Tupla modificada:", t_modificada)
```

Solución 11

```
punto = (4, 7)

x, y = punto

print("x =", x)
print("y =", y)
```

Solución 12

```
t = (1, 2, 3, 4, 5)

primero, *medio, ultimo = t
```

```
print("Primero:", primero)
print("Medio:", medio)
print("Último:", ultimo)
```

Solución 13

```
matriz = (
    (1, 2),
    (3, 4)
)

# fila 2, columna 1 => matriz[1][0]
elemento = matriz[1][0]
print("Elemento fila 2, columna 1:", elemento)
```

Solución 14

```
matriz = (
    (1, 2),
    (3, 4)
)

for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print()
```

Solución 15

```
poblacion = {
    ("España", "Zaragoza"): 675301,
    ("España", "Madrid"): 3223334,
    ("Francia", "París"): 2148327
}
```

```
print("Población de Zaragoza:", poblacion[("España", "Zaragoza")])
```

Solución 16

```
def suma_y_producto(a, b):
    suma = a + b
    producto = a * b
    return suma, producto

s, p = suma_y_producto(3, 4)

print("Suma:", s)
print("Producto:", p)
```

Solución 17

```
pares = [(1, 2), (3, 4), (5, 6)]

for a, b in pares:
    print(f"{a} + {b} = {a + b}")
```

Solución 18

```
t = (1, 2, 3)

# t.append(4)   # AttributeError: 'tuple' object has no attribute 'append'
# t.remove(2)   # AttributeError: 'tuple' object has no attribute 'remove'

print("Las tuplas no tienen métodos append ni remove porque son inmutables.")
```

Solución 19

```
t = ([1, 2], [3, 4])
```

```
t[0][0] = 99

print("Tupla después de modificar la lista interna:", t)
```

Explicación: la tupla es inmutable (no podemos cambiar qué listas contiene), pero las **listas internas sí son mutables**, por eso podemos cambiar sus elementos.

Solución 20

```
frase = input("Introduce una frase: ")

palabras = tuple(frase.split())

print("Tupla de palabras:", palabras)
print("Número de palabras:", len(palabras))

if len(palabras) > 0:
    print("Primera palabra:", palabras[0])
    print("Última palabra:", palabras[-1])
```

---

9. Diccionarios en Python

Los **diccionarios** son una de las estructuras de datos más poderosas y utilizadas en Python. Mientras que las listas, tuplas y conjuntos trabajan con **colecciones de valores**, los diccionarios trabajan con **pares clave–valor**: cada valor está asociado a una **clave** que lo identifica.

Un diccionario es ideal cuando quieres **buscar información por un identificador**: por ejemplo, buscar el teléfono de una persona por su nombre, el precio de un producto por su código, las notas de un alumno por su DNI, etc.

En esta lección veremos:

- **9.1 Creación de diccionarios**
- **9.2 Acceso a claves y valores**
- **9.3 Métodos fundamentales**
- **9.4 Diccionarios anidados**
- **9.5 Iteración sobre diccionarios**
- **9.6 Errores comunes**

Al final tendrás **20 ejercicios** con su **solución detallada**.

9.1 Creación de diccionarios

Un **diccionario** en Python es una colección **no ordenada** (en versiones modernas mantiene el orden de inserción, pero conceptualmente no debes depender de ello), **mutable** y que almacena pares **clave–valor**.

Se define usando **llaves** `{}` y separando cada par con comas.
Cada par se escribe como `clave: valor`.

9.1.1 Ejemplos básicos de creación

```
vacio = {}  # diccionario vacío


edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}


precios = {
    "manzana": 0.5,
    "pera": 0.75,
    "plátano": 0.6
}
```

```
print(vacio)
print(edades)
print(precios)
```

En `edades`, las **claves** son cadenas (`"Ana"`, `"Luis"`, `"María"`) y los **valores** son enteros (las edades).
En `precios`, las claves son nombres de frutas y los valores son precios.

9.1.2 Claves y tipos permitidos

Las **claves** de un diccionario deben ser **hashables**, es decir, tipos inmutables:
cadenas, números, tuplas inmutables, frozenset, etc.

No se pueden usar listas ni otros diccionarios mutables como claves.

```
dic = {
    1: "uno",
    "dos": 2,
    (3, 4): "coordenada"
}

print(dic[1])        # "uno"
print(dic["dos"])    # 2
print(dic[(3, 4)])   # "coordenada"
```

9.1.3 Crear diccionarios con dict()

También podemos usar la función **dict()**:

```
persona = dict(nombre="Ana", edad=25, ciudad="Zaragoza")
print(persona)
```

Aquí las claves deben ser identificadores válidos (sin espacios, sin caracteres especiales).

También se puede crear a partir de una lista de tuplas:

```
lista_pares = [("uno", 1), ("dos", 2), ("tres", 3)]
numeros = dict(lista_pares)

print(numeros)  # {'uno': 1, 'dos': 2, 'tres': 3}
```

9.1.4 Diccionario vacío y añadir después

```
datos = {}
```

```
datos["nombre"] = "Luis"
datos["edad"] = 30
datos["ciudad"] = "Zaragoza"

print(datos)
```

9.2 Acceso a claves y valores

Una vez creado un diccionario, queremos **consultar** y **modificar** sus valores.

9.2.1 Acceso por clave

La forma básica es:

```
valor = diccionario[clave]
```

Ejemplo:

```
edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}

print(edades["Ana"])   # 25
print(edades["Luis"])  # 30
```

Si la clave **no existe**, se produce un **KeyError**:

```
# print(edades["Pedro"])  # KeyError: 'Pedro'
```

9.2.2 Modificación de valores

Podemos cambiar el valor asociado a una clave:

```
edades["Ana"] = 26
print(edades["Ana"])  # 26
```

Si la clave no existe, se **crea**:

```
edades["Pedro"] = 28
print(edades)
```

9.2.3 Comprobar si una clave existe

Usamos el operador `in` sobre las **claves**:

```
edades = {"Ana": 25, "Luis": 30}
```

```
print("Ana" in edades)    # True
print("Pedro" in edades)  # False
```

9.2.4 Acceso seguro con get()

El método **get()** permite acceder a una clave sin provocar error si no existe:

```
edades = {"Ana": 25, "Luis": 30}

print(edades.get("Ana"))      # 25
print(edades.get("Pedro"))    # None
print(edades.get("Pedro", 0)) # 0 (valor por defecto)
```

Ventaja: evita el `KeyError` y permite especificar un valor por defecto.

9.3 Métodos fundamentales

Los diccionarios tienen muchos métodos útiles.
Los más importantes:

- **keys()** → devuelve una vista de las claves.
- **values()** → devuelve una vista de los valores.
- **items()** → devuelve una vista de pares (clave, valor).
- **get(clave, defecto)** → acceso seguro.
- **update(otro_dic)** → actualiza con pares de otro diccionario.
- **pop(clave[, defecto])** → elimina y devuelve el valor de una clave.
- **popitem()** → elimina y devuelve el último par insertado.
- **clear()** → vacía el diccionario.
- **copy()** → copia superficial.

9.3.1 keys(), values(), items()

```
persona = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Zaragoza"
}

print(persona.keys())    # dict_keys(['nombre', 'edad', 'ciudad'])
print(persona.values())  # dict_values(['Ana', 25, 'Zaragoza'])
print(persona.items())   # dict_items([('nombre', 'Ana'), ('edad',
25), ('ciudad', 'Zaragoza')])
```

Podemos convertir estas vistas en listas:

```
claves = list(persona.keys())
valores = list(persona.values())
pares = list(persona.items())

print(claves)
print(valores)
print(pares)
```

9.3.2 update()

`update()` permite **fusionar** diccionarios o actualizar valores:

```
config = {
    "color": "azul",
    "tamaño": "mediano"
}

nueva_config = {
    "tamaño": "grande",
    "fuente": "Arial"
}

config.update(nueva_config)

print(config)
# {'color': 'azul', 'tamaño': 'grande', 'fuente': 'Arial'}
```

Las claves existentes se **sobrescriben**, las nuevas se añaden.

9.3.3 pop()

`pop(clave)` elimina la clave y devuelve su valor:

```
edades = {"Ana": 25, "Luis": 30, "María": 22}

edad_ana = edades.pop("Ana")
print("Edad de Ana:", edad_ana)
print("Diccionario:", edades)
```

Si la clave no existe y no se da valor por defecto, lanza `KeyError`:

```
# edades.pop("Pedro")  # KeyError
```

Con valor por defecto:

```
valor = edades.pop("Pedro", None)
print(valor)  # None
```

9.3.4 popitem()

`popitem()` elimina y devuelve el **último par insertado** (en versiones modernas de Python):

```
d = {"a": 1, "b": 2, "c": 3}

clave, valor = d.popitem()
print("Eliminado:", clave, valor)
print("Diccionario:", d)
```

9.3.5 clear() y copy()

```
d = {"a": 1, "b": 2}
copia = d.copy()

d.clear()

print("Original:", d)  # {}
print("Copia:", copia) # {'a': 1, 'b': 2}
```

9.4 Diccionarios anidados

Un **diccionario anidado** es un diccionario que contiene otros diccionarios como valores.
Se usan para representar estructuras más complejas: por ejemplo, una base de datos en memoria.

9.4.1 Ejemplo de diccionario anidado

```
alumnos = {
    "A001": {
        "nombre": "Ana",
        "edad": 20,
        "notas": [8, 9, 7]
    },
    "A002": {
```

```
        "nombre": "Luis",
        "edad": 22,
        "notas": [6, 7, 8]
    }
}

print(alumnos["A001"]["nombre"])  # Ana
print(alumnos["A002"]["notas"])   # [6, 7, 8]
```

9.4.2 Añadir un nuevo alumno

```
alumnos["A003"] = {
    "nombre": "María",
    "edad": 21,
    "notas": [9, 9, 10]
}

print(alumnos["A003"])
```

9.4.3 Modificar datos internos

```
alumnos["A001"]["edad"] = 21
alumnos["A002"]["notas"].append(9)

print(alumnos["A001"])
print(alumnos["A002"])
```

9.5 Iteración sobre diccionarios

Recorrer diccionarios es una operación muy habitual.

9.5.1 Recorrer claves

```
edades = {"Ana": 25, "Luis": 30, "María": 22}

for nombre in edades:
    print(nombre, "tiene", edades[nombre], "años")
```

Equivalente a:

```
for nombre in edades.keys():
```

```
    print(nombre, "tiene", edades[nombre], "años")
```

9.5.2 Recorrer valores

```
for edad in edades.values():
    print("Edad:", edad)
```

9.5.3 Recorrer pares clave–valor

```
for nombre, edad in edades.items():
    print(nombre, "=>", edad)
```

Esto es muy cómodo y legible.

9.5.4 Ejemplo práctico: sumar valores

```
notas = {
    "Ana": 8.5,
    "Luis": 7.0,
    "María": 9.2
}

suma = 0
for alumno, nota in notas.items():
    print(alumno, "tiene nota", nota)
    suma += nota

media = suma / len(notas)
print("Nota media:", media)
```

9.6 Errores comunes

Trabajando con diccionarios es fácil caer en algunos errores típicos.

9.6.1 Acceder a una clave inexistente

```
edades = {"Ana": 25, "Luis": 30}

# print(edades["Pedro"])  # KeyError
```

Soluciones:

- Comprobar antes con `in`:

```
if "Pedro" in edades:
```

```
    print(edades["Pedro"])
else:
    print("Pedro no está en el diccionario")
```

- Usar `get()` con valor por defecto:

```
print(edades.get("Pedro", "Desconocido"))
```

9.6.2 Confundir claves y valores

```
edades = {"Ana": 25, "Luis": 30}

print(25 in edades)          # False (busca en claves)
print("Ana" in edades)       # True
print(25 in edades.values()) # True
```

Recuerda: `in diccionario` comprueba en las **claves**, no en los valores.

9.6.3 Modificar el diccionario mientras se itera

```
d = {"a": 1, "b": 2, "c": 3}

# for clave in d:
#     if clave == "b":
#         del d[clave]  # puede causar problemas
```

Modificar la estructura mientras se recorre puede producir errores o comportamientos inesperados.

Solución: crear una lista de claves a eliminar:

```
d = {"a": 1, "b": 2, "c": 3}
a_eliminar = [clave for clave, valor in d.items() if valor % 2 == 0]

for clave in a_eliminar:
    del d[clave]

print(d)  # {'a': 1, 'c': 3}
```

9.6.4 Usar tipos no hashables como claves

```
# d = {[1, 2]: "lista"}  # TypeError: unhashable type: 'list'
```

Las claves deben ser inmutables: usa tuplas si necesitas agrupar datos.

```
d = {(1, 2): "coordenada"}
```

```
print(d[(1, 2)])
```

9.6.5 Pensar que copy() hace copia profunda

```
original = {
    "a": [1, 2],
    "b": [3, 4]
}

copia = original.copy()
copia["a"][0] = 99

print(original)  # {'a': [99, 2], 'b': [3, 4]}
print(copia)     # {'a': [99, 2], 'b': [3, 4]}
```

`copy()` hace una **copia superficial**: las listas internas se comparten.
Si necesitas copia profunda, usa `copy.deepcopy()`.

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Crea un diccionario con las edades de tres personas: `"Ana": 25`, `"Luis": 30`, `"María": 22`. Muestra el diccionario y el tipo de dato.

Ejercicio 2
Accede y muestra la edad de `"Luis"` del diccionario anterior.

Ejercicio 3
Añade una nueva persona `"Pedro": 28` al diccionario de edades y muestra el diccionario actualizado.

Ejercicio 4
Modifica la edad de `"Ana"` a 26 en el diccionario y muestra el resultado.

Ejercicio 5
Comprueba si `"María"` está en el diccionario de edades. Si está, muestra su edad; si no, muestra un mensaje indicando que no está.

Ejercicio 6
Usa `get()` para obtener la edad de `"Carlos"` del diccionario de edades, devolviendo 0 si no existe.

Ejercicio 7
Crea un diccionario `precios` con al menos 3 productos y sus precios. Muestra la lista de claves, la lista de valores y la lista de pares clave–valor.

Ejercicio 8
Usa `update()` para añadir o modificar varios productos en el diccionario `precios` de una sola vez.

Ejercicio 9
Usa `pop()` para eliminar un producto del diccionario `precios` y muestra el precio eliminado y el diccionario resultante.

Ejercicio 10
Usa `popitem()` para eliminar el último par insertado en el diccionario `precios` y muestra el par eliminado.

Ejercicio 11
Crea un diccionario `alumno` con claves `"nombre"`, `"edad"` y `"notas"` (siendo `"notas"` una lista de números). Muestra el nombre del alumno y su nota media.

Ejercicio 12
Crea un diccionario `alumnos` donde las claves sean códigos de alumno (por ejemplo `"A001"`, `"A002"`) y los valores sean diccionarios con `"nombre"` y `"edad"`. Añade al menos 2 alumnos y muestra sus datos.

Ejercicio 13
En el diccionario `alumnos` del ejercicio anterior, añade para cada alumno una lista de notas bajo la clave `"notas"`. Muestra las notas de un alumno concreto.

Ejercicio 14
Crea un diccionario `notas` con nombres de alumnos y sus notas numéricas. Recorre el diccionario con `items()` y muestra el nombre y la nota de cada alumno.

Ejercicio 15
Con el diccionario `notas` del ejercicio anterior, calcula la nota media de la clase.

Ejercicio 16
Crea un diccionario que use como clave una tupla `(pais, ciudad)` y como valor la población. Añade al menos 3 entradas y muestra todas las claves y valores.

Ejercicio 17
Dado un diccionario de productos y precios, pide al usuario el nombre de un producto y muestra su precio si existe, o un mensaje si no existe.

Ejercicio 18
Crea un diccionario con varias claves y valores. Crea una copia superficial con `copy()` y demuestra que si modificas una lista interna en la copia, también cambia en el original.

Ejercicio 19
Escribe un programa que cuente cuántas veces aparece cada palabra en una frase introducida por el usuario, usando un diccionario donde la clave sea la palabra y el valor sea la frecuencia.

Ejercicio 20
Escribe un programa que pida al usuario nombres de alumnos y sus notas (hasta que introduzca una cadena vacía como nombre), los guarde en un diccionario y luego muestre todos los alumnos con su nota.

Soluciones de los 20 ejercicios

Solución 1

```
edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}

print("Diccionario de edades:", edades)
print("Tipo:", type(edades))
```

Solución 2

```
edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}

print("Edad de Luis:", edades["Luis"])
```

Solución 3

```
edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}

edades["Pedro"] = 28
```

```
print("Diccionario actualizado:", edades)
```

Solución 4

```
edades = {
    "Ana": 25,
    "Luis": 30,
    "María": 22
}

edades["Ana"] = 26

print("Diccionario con la nueva edad de Ana:", edades)
```

Solución 5

```
edades = {
    "Ana": 26,
    "Luis": 30,
    "María": 22
}

if "María" in edades:
    print("La edad de María es:", edades["María"])
else:
    print("María no está en el diccionario.")
```

Solución 6

```
edades = {
    "Ana": 26,
    "Luis": 30,
    "María": 22
}
```

```
edad_carlos = edades.get("Carlos", 0)
print("Edad de Carlos (0 si no existe):", edad_carlos)
```

Solución 7

```
precios = {
    "manzana": 0.5,
    "pera": 0.75,
    "plátano": 0.6
}

claves = list(precios.keys())
valores = list(precios.values())
pares = list(precios.items())

print("Claves:", claves)
print("Valores:", valores)
print("Pares clave-valor:", pares)
```

Solución 8

```
precios = {
    "manzana": 0.5,
    "pera": 0.75,
    "plátano": 0.6
}

nuevos_precios = {
    "pera": 0.8,
    "naranja": 0.9
}

precios.update(nuevos_precios)
```

```
print("Diccionario de precios actualizado:", precios)
```

Solución 9

```
precios = {
    "manzana": 0.5,
    "pera": 0.8,
    "plátano": 0.6,
    "naranja": 0.9
}

precio_pera = precios.pop("pera")
print("Precio eliminado de la pera:", precio_pera)
print("Diccionario de precios:", precios)
```

Solución 10

```
precios = {
    "manzana": 0.5,
    "plátano": 0.6,
    "naranja": 0.9
}

clave, valor = precios.popitem()
print("Par eliminado:", clave, "=>", valor)
print("Diccionario restante:", precios)
```

Solución 11

```
alumno = {
    "nombre": "Ana",
    "edad": 20,
    "notas": [8, 9, 7]
}
```

```
nombre = alumno["nombre"]
notas = alumno["notas"]
media = sum(notas) / len(notas)

print("Nombre:", nombre)
print("Notas:", notas)
print("Nota media:", media)
```

Solución 12

```
alumnos = {
    "A001": {
        "nombre": "Ana",
        "edad": 20
    },
    "A002": {
        "nombre": "Luis",
        "edad": 22
    }
}

for codigo, datos in alumnos.items():
    print("Código:", codigo)
    print("  Nombre:", datos["nombre"])
    print("  Edad:", datos["edad"])
```

Solución 13

```
alumnos = {
    "A001": {
        "nombre": "Ana",
        "edad": 20,
        "notas": [8, 9, 7]
```

```
    },
    "A002": {
        "nombre": "Luis",
        "edad": 22,
        "notas": [6, 7, 8]
    }
}

codigo = "A001"
print("Notas del alumno", codigo, ":", alumnos[codigo]["notas"])
```

Solución 14

```
notas = {
    "Ana": 8.5,
    "Luis": 7.0,
    "María": 9.2
}

for nombre, nota in notas.items():
    print(nombre, "tiene nota", nota)
```

Solución 15

```
notas = {
    "Ana": 8.5,
    "Luis": 7.0,
    "María": 9.2
}

suma = 0
for nota in notas.values():
    suma += nota
```

```
media = suma / len(notas)
print("Nota media de la clase:", media)
```

Solución 16

```
poblacion = {
    ("España", "Zaragoza"): 675301,
    ("España", "Madrid"): 3223334,
    ("Francia", "París"): 2148327
}

for clave, valor in poblacion.items():
    pais, ciudad = clave
    print(f"{ciudad}, {pais}: {valor} habitantes")
```

Solución 17

```
precios = {
    "manzana": 0.5,
    "pera": 0.8,
    "plátano": 0.6
}

producto = input("Introduce el nombre de un producto: ")

if producto in precios:
    print("El precio de", producto, "es", precios[producto])
else:
    print("El producto no existe en la lista de precios.")
```

Solución 18

```
import copy

original = {
```

```
    "a": [1, 2],
    "b": [3, 4]
}

copia = original.copy()

copia["a"][0] = 99

print("Original:", original)
print("Copia:", copia)
```

Explicación: `copy()` hace una copia superficial; las listas internas se comparten entre `original` y `copia`.

Solución 19

```
frase = input("Introduce una frase: ")

palabras = frase.split()
frecuencias = {}

for palabra in palabras:
    palabra = palabra.lower()
    if palabra in frecuencias:
        frecuencias[palabra] += 1
    else:
        frecuencias[palabra] = 1

print("Frecuencia de palabras:")
for palabra, cuenta in frecuencias.items():
    print(palabra, "=>", cuenta)
```

Solución 20

```
alumnos = {}
```

```
while True:
    nombre = input("Introduce el nombre del alumno (o vacío para terminar): ")
    if nombre == "":
        break
    nota = float(input(f"Introduce la nota de {nombre}: "))
    alumnos[nombre] = nota

print("Alumnos y sus notas:")
for nombre, nota in alumnos.items():
    print(nombre, "=>", nota)
```

---

10. Funciones en Python

Las **funciones** son uno de los pilares de la programación. Permiten **organizar el código**, **reutilizarlo**, evitar repeticiones, dividir problemas grandes en partes pequeñas y hacer que los programas sean más claros y fáciles de mantener.

En Python, trabajar bien con funciones es imprescindible si quieres escribir programas medianos o grandes, o simplemente código limpio y profesional.

En esta lección veremos, con teoría muy detallada y muchos ejemplos:

- **10.1 Definición y llamada**
- **10.2 Parámetros y argumentos**
- **10.3 Valores por defecto**
- **10.4 args y **kwargs*
- **10.5 Funciones lambda**
- **10.6 Ámbito de variables**
- **10.7 Recursividad**
- **10.8 Buenas prácticas**

Al final tendrás **20 ejercicios** con su **solución detallada**.

10.1 Definición y llamada

Una **función** es un bloque de código con un **nombre**, que puede recibir **datos de entrada** (parámetros), realizar operaciones y opcionalmente **devolver un resultado**.

En Python se define con la palabra clave **`def`**.

10.1.1 Sintaxis básica

```
def nombre_de_la_funcion(parámetros_opcionales):
    # bloque de código
    # ...
    return valor_opcional
```

- **`def`**: palabra clave para definir funciones.
- **`nombre_de_la_funcion`**: identificador (como el nombre de una variable).
- **`parámetros_opcionales`**: cero o más nombres de parámetros.
- **`return`**: devuelve un valor (opcional).

10.1.2 Ejemplo simple sin parámetros ni retorno

```
def saludar():
    print("Hola, bienvenido a Python")


# Llamada a la función
saludar()
```

```
saludar()
```

Salida:

```
Hola, bienvenido a Python
Hola, bienvenido a Python
```

La función se define una vez y se puede **llamar** todas las veces que quieras.

10.1.3 Función con parámetros

```
def saludar_persona(nombre):
    print("Hola,", nombre)

saludar_persona("Ana")
saludar_persona("Luis")
```

Salida:

```
Hola, Ana
Hola, Luis
```

10.1.4 Función con valor de retorno

```
def cuadrado(n):
    return n * n

resultado = cuadrado(5)
print("El cuadrado de 5 es", resultado)
```

Salida:

```
El cuadrado de 5 es 25
```

Si una función no tiene `return`, devuelve implícitamente `None`.

10.2 Parámetros y argumentos

Parámetros: nombres que aparecen en la definición de la función.
Argumentos: valores concretos que pasamos al llamar a la función.

```
def sumar(a, b):   # a y b son parámetros
    return a + b

resultado = sumar(3, 4)  # 3 y 4 son argumentos
print(resultado)
```

10.2.1 Parámetros posicionales

Los argumentos se asocian a los parámetros **por posición**:

```
def presentar(nombre, edad):
    print("Nombre:", nombre)
    print("Edad:", edad)


presentar("Ana", 25)   # nombre="Ana", edad=25
presentar("Luis", 30)  # nombre="Luis", edad=30
```

10.2.2 Argumentos nombrados (keyword arguments)

Podemos indicar explícitamente qué argumento va a qué parámetro:

```
presentar(edad=40, nombre="María")
```

Esto mejora la legibilidad y permite cambiar el orden.

10.2.3 Mezcla de posicionales y nombrados

```
def mostrar_datos(nombre, edad, ciudad):
    print("Nombre:", nombre)
    print("Edad:", edad)
    print("Ciudad:", ciudad)


mostrar_datos("Ana", 25, "Zaragoza")                # solo
posicionales
mostrar_datos("Luis", ciudad="Madrid", edad=30)      # mezcla
mostrar_datos(nombre="María", edad=22, ciudad="Sevilla")  # todos
nombrados
```

Regla importante: primero argumentos posicionales, luego nombrados.

```
# mostrar_datos(nombre="Ana", 25, "Zaragoza")  # ERROR
```

10.3 Valores por defecto

Podemos definir parámetros con **valores por defecto**.
Si el argumento no se pasa, se usa el valor por defecto.

10.3.1 Ejemplo básico

```
def saludar(nombre="invitado"):
    print("Hola,", nombre)
```

```
saludar("Ana")      # Hola, Ana
saludar()           # Hola, invitado
```

10.3.2 Varios parámetros con valores por defecto

```
def potencia(base, exponente=2):
    return base ** exponente

print(potencia(3))       # 9 (3^2)
print(potencia(3, 3))    # 27 (3^3)
```

10.3.3 Orden de parámetros

Los parámetros con valor por defecto deben ir **después** de los que no tienen:

```
def ejemplo(a, b=10):
    print(a, b)

# def mal(a=10, b):  # ERROR
#     ...
```

10.4 *args y **kwargs

A veces no sabemos cuántos argumentos va a recibir una función.
Para eso existen:

- **`*args`**: argumentos posicionales variables.
- **`**kwargs`**: argumentos nombrados variables.

Los nombres `args` y `kwargs` son una convención, pero lo importante son los símbolos `*` y `**`.

10.4.1 *args: número variable de argumentos posicionales

```
def sumar_todos(*numeros):
    total = 0
    for n in numeros:
        total += n
    return total

print(sumar_todos(1, 2, 3))          # 6
print(sumar_todos(10, 20, 30, 40))   # 100
print(sumar_todos())                 # 0
```

Dentro de la función, `numeros` es una **tupla** con todos los argumentos posicionales.

10.4.2 **kwargs: número variable de argumentos nombrados

```
def mostrar_info(**datos):
    for clave, valor in datos.items():
        print(clave, "=>", valor)

mostrar_info(nombre="Ana", edad=25, ciudad="Zaragoza")
```

Salida:

```
nombre => Ana
edad => 25
ciudad => Zaragoza
```

Dentro de la función, `datos` es un **diccionario**.

10.4.3 Combinar parámetros normales, *args y **kwargs

```
def demo(a, b, *args, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("kwargs:", kwargs)

demo(1, 2, 3, 4, 5, x=10, y=20)
```

Salida:

```
a: 1
b: 2
args: (3, 4, 5)
kwargs: {'x': 10, 'y': 20}
```

Orden recomendado de parámetros:

1. Parámetros posicionales normales
2. Parámetros con valor por defecto
3. `*args`
4. `**kwargs`

10.5 Funciones lambda

Las **funciones lambda** son funciones **anónimas** y **pequeñas**, definidas en una sola línea.
Se usan mucho cuando necesitamos una función “rápida” como argumento de otra función.

10.5.1 Sintaxis

```
lambda parámetros: expresión
```

Ejemplo:

```
cuadrado = lambda x: x * x


print(cuadrado(5))  # 25
```

Equivalente a:

```
def cuadrado(x):
    return x * x
```

10.5.2 Lambda con varios parámetros

```
sumar = lambda a, b: a + b
print(sumar(3, 4))  # 7
```

10.5.3 Uso típico con sorted, map, filter

Ordenar una lista de tuplas por el segundo elemento:

```
datos = [("Ana", 25), ("Luis", 30), ("María", 22)]


ordenados = sorted(datos, key=lambda x: x[1])
print(ordenados)  # [('María', 22), ('Ana', 25), ('Luis', 30)]
```

Filtrar números pares:

```
numeros = [1, 2, 3, 4, 5, 6]


pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # [2, 4, 6]
```

Aplicar una operación a cada elemento (map):

```
numeros = [1, 2, 3, 4]


cuadrados = list(map(lambda x: x * x, numeros))
print(cuadrados)  # [1, 4, 9, 16]
```

10.6 Ámbito de variables

El **ámbito** (scope) de una variable indica **dónde es visible** y **dónde existe**.

- **Variables locales**: definidas dentro de una función.
- **Variables globales**: definidas fuera de cualquier función.

10.6.1 Variables locales

```
def ejemplo():
    x = 10  # variable local
    print("Dentro de la función, x =", x)

ejemplo()
# print(x)  # ERROR: x no existe fuera de la función
```

`x` solo existe dentro de `ejemplo`.

10.6.2 Variables globales

```
x = 100  # variable global

def mostrar():
    print("Dentro de la función, x =", x)

mostrar()
print("Fuera de la función, x =", x)
```

10.6.3 Sombra de variables (shadowing)

```
x = 100

def ejemplo():
    x = 50  # variable local, distinta de la global
    print("Dentro de la función, x =", x)

ejemplo()
print("Fuera de la función, x =", x)
```

Salida:

```
Dentro de la función, x = 50
Fuera de la función, x = 100
```

La variable local **oculta** a la global dentro de la función.

10.6.4 Modificar una global dentro de una función: global

```
contador = 0

def incrementar():
    global contador
    contador += 1

incrementar()
incrementar()
print("Contador:", contador)  # 2
```

Usar `global` debe hacerse con cuidado; abusar de variables globales hace el código difícil de mantener.

10.7 Recursividad

Una función es **recursiva** cuando **se llama a sí misma**.
La recursividad es útil para problemas que se pueden definir en términos de versiones más pequeñas de sí mismos (factorial, Fibonacci, recorrer estructuras anidadas, etc.).

10.7.1 Estructura de una función recursiva

1. **Caso base**: condición que detiene la recursión.
2. **Caso recursivo**: la función se llama a sí misma con un problema más pequeño.

10.7.2 Ejemplo: factorial

El factorial de un número entero positivo `n` se define como:

- `0! = 1`
- `n! = n * (n-1)!` para `n > 0`

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # 120
```

10.7.3 Ejemplo: suma de 1 a n

```
def suma_hasta(n):
    if n == 0:
        return 0
    else:
        return n + suma_hasta(n - 1)

print(suma_hasta(5))  # 15
```

10.7.4 Ejemplo: Fibonacci

Secuencia de Fibonacci:

- `f(0) = 0`
- `f(1) = 1`
- `f(n) = f(n-1) + f(n-2)` para `n >= 2`

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))  # 8
```

Nota: esta versión es correcta pero ineficiente para valores grandes.

10.8 Buenas prácticas

Al trabajar con funciones, conviene seguir ciertas **buenas prácticas** para que el código sea más claro, mantenible y profesional.

10.8.1 Nombres descriptivos

- Usa nombres que describan lo que hace la función: `calcular_media`, `obtener_maximo`, `es_primo`, etc.
- Evita nombres genéricos como `funcion1`, `hacer_algo`.

10.8.2 Una función, una responsabilidad

Cada función debería hacer **una cosa concreta**.
Si una función hace demasiadas cosas, conviene dividirla en varias.

10.8.3 Documentar con docstrings

Usa **docstrings** (cadenas de documentación) para explicar qué hace la función, qué parámetros recibe y qué devuelve.

```
def area_circulo(radio):
    """
    Calcula el área de un círculo dado su radio.

    Parámetros:
        radio (float): radio del círculo.

    Devuelve:
        float: área del círculo.
    """
    import math
    return math.pi * radio ** 2
```

Puedes ver la documentación con `help(area_circulo)`.

10.8.4 Evitar abusar de variables globales

Las funciones deberían depender principalmente de sus **parámetros** y **valores de retorno**, no de variables globales.

10.8.5 Mantener las funciones cortas

Si una función es muy larga, probablemente se pueda dividir en funciones más pequeñas y reutilizables.

---

20 Ejercicios propuestos

No mires las soluciones hasta intentar resolverlos tú mismo.

Ejercicio 1
Escribe una función `saludar()` que no reciba parámetros y muestre por pantalla un saludo simple. Llámala varias veces.

Ejercicio 2
Escribe una función `saludar_persona(nombre)` que reciba un nombre y muestre un saludo personalizado.

Ejercicio 3
Escribe una función `sumar(a, b)` que reciba dos números y devuelva su suma. Muestra el resultado de llamar a la función.

Ejercicio 4
Escribe una función `es_par(n)` que devuelva `True` si `n` es par y `False` en caso contrario. Pruébala con varios valores.

Ejercicio 5
Escribe una función `potencia(base, exponente=2)` que devuelva `base` elevado a `exponente`, con valor por defecto 2. Pruébala con uno y con dos argumentos.

Ejercicio 6
Escribe una función `presentar(nombre, edad=0, ciudad="desconocida")` que muestre un mensaje con esos datos. Pruébala con distintos argumentos, usando posicionales y nombrados.

Ejercicio 7
Escribe una función `sumar_todos(*numeros)` que reciba un número variable de argumentos y devuelva la suma de todos ellos. Pruébala con diferentes cantidades de números.

Ejercicio 8
Escribe una función `mostrar_info(**datos)` que reciba un número variable de argumentos nombrados y los muestre por pantalla.

Ejercicio 9
Escribe una función `operar(a, b, *args, **kwargs)` que muestre los valores de `a`, `b`, `args` y `kwargs` para entender cómo se reciben.

Ejercicio 10
Escribe una función lambda que reciba un número y devuelva su cuadrado. Úsala para calcular el cuadrado de 5 y 10.

Ejercicio 11
Usa una función lambda y `sorted()` para ordenar una lista de tuplas `(nombre, edad)` por edad.

Ejercicio 12
Escribe una función `contar_vocales(texto)` que devuelva cuántas vocales hay en una cadena. Usa variables locales dentro de la función.

Ejercicio 13
Declara una variable global `contador = 0` y una función `incrementar()` que aumente en 1 el valor de `contador` usando `global`. Llama varias veces a la función y muestra el valor final de `contador`.

Ejercicio 14
Escribe una función recursiva `factorial(n)` que calcule el factorial de `n`. Pruébala con varios valores.

Ejercicio 15
Escribe una función recursiva `suma_hasta(n)` que devuelva la suma de los números de 1 a `n`. Pruébala con `n = 5` y `n = 10`.

Ejercicio 16
Escribe una función recursiva `contar_atras(n)` que muestre una cuenta atrás desde `n` hasta 0.

Ejercicio 17
Escribe una función `es_palindromo(texto)` que devuelva `True` si `texto` es un palíndromo (se lee igual al derecho y al revés, ignorando espacios y mayúsculas) y `False` en caso contrario.

Ejercicio 18
Escribe una función `maximo(a, b, c)` que devuelva el mayor de tres números. No uses `max`, hazlo con condicionales.

Ejercicio 19
Escribe una función `media(lista)` que reciba una lista de números y devuelva su media. Si la lista está vacía, devuelve `None`.

Ejercicio 20
Escribe una función bien documentada con docstring, por ejemplo `area_rectangulo(base, altura)`, que devuelva el área de un rectángulo. Muestra su docstring con `help()`.

Soluciones de los 20 ejercicios

Solución 1

```
def saludar():
    print("Hola, bienvenido a Python")

saludar()
saludar()
saludar()
```

Solución 2

```
def saludar_persona(nombre):
    print("Hola,", nombre)

saludar_persona("Ana")
saludar_persona("Luis")
```

Solución 3

```
def sumar(a, b):
    return a + b
```

```
resultado = sumar(3, 4)
print("La suma es:", resultado)
```

Solución 4

```
def es_par(n):
    return n % 2 == 0

print(es_par(4))  # True
print(es_par(7))  # False
```

Solución 5

```
def potencia(base, exponente=2):
    return base ** exponente

print(potencia(3))      # 9
print(potencia(3, 3))   # 27
```

Solución 6

```
def presentar(nombre, edad=0, ciudad="desconocida"):
    print("Nombre:", nombre)
    print("Edad:", edad)
    print("Ciudad:", ciudad)
    print("-----")

presentar("Ana")
presentar("Luis", 30)
presentar("María", 22, "Zaragoza")
presentar(nombre="Pedro", ciudad="Madrid", edad=40)
```

Solución 7

```
def sumar_todos(*numeros):
```

```
    total = 0
    for n in numeros:
        total += n
    return total

print(sumar_todos(1, 2, 3))          # 6
print(sumar_todos(10, 20, 30, 40))   # 100
print(sumar_todos())                 # 0
```

Solución 8

```
def mostrar_info(**datos):
    for clave, valor in datos.items():
        print(clave, "=>", valor)

mostrar_info(nombre="Ana", edad=25, ciudad="Zaragoza")
mostrar_info(producto="Manzana", precio=0.5, stock=100)
```

Solución 9

```
def operar(a, b, *args, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("kwargs:", kwargs)

operar(1, 2)
operar(1, 2, 3, 4, x=10, y=20)
```

Solución 10

```
cuadrado = lambda x: x * x

print(cuadrado(5))   # 25
print(cuadrado(10))  # 100
```

Solución 11

```
personas = [("Ana", 25), ("Luis", 30), ("María", 22)]

ordenadas = sorted(personas, key=lambda p: p[1])

print(ordenadas)  # [('María', 22), ('Ana', 25), ('Luis', 30)]
```

Solución 12

```
def contar_vocales(texto):
    texto = texto.lower()
    vocales = "aeiouáéíóúü"
    contador = 0
    for c in texto:
        if c in vocales:
            contador += 1
    return contador

frase = "Hola, ¿qué tal?"
print("Número de vocales:", contar_vocales(frase))
```

Solución 13

```
contador = 0

def incrementar():
    global contador
    contador += 1

incrementar()
incrementar()
incrementar()
```

```
print("Valor de contador:", contador)  # 3
```

Solución 14

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # 120
print(factorial(0))  # 1
```

Solución 15

```
def suma_hasta(n):
    if n == 0:
        return 0
    else:
        return n + suma_hasta(n - 1)

print(suma_hasta(5))   # 15
print(suma_hasta(10))  # 55
```

Solución 16

```
def contar_atras(n):
    if n < 0:
        return
    print(n)
    contar_atras(n - 1)

contar_atras(5)
```

Salida:

```
5
```

```
4
3
2
1
0
```

Solución 17

```
def es_palindromo(texto):
    texto = texto.lower().replace(" ", "")
    return texto == texto[::-1]

print(es_palindromo("Ana"))             # True
print(es_palindromo("Anita lava la tina"))  # True
print(es_palindromo("Hola"))           # False
```

Solución 18

```
def maximo(a, b, c):
    mayor = a
    if b > mayor:
        mayor = b
    if c > mayor:
        mayor = c
    return mayor

print(maximo(3, 7, 5))   # 7
print(maximo(10, 2, 8))  # 10
```

Solución 19

```
def media(lista):
    if len(lista) == 0:
        return None
    return sum(lista) / len(lista)
```

```
print(media([1, 2, 3, 4, 5]))  # 3.0
print(media([]))               # None
```

Solución 20

```
def area_rectangulo(base, altura):
    """
    Calcula el área de un rectángulo.

    Parámetros:
        base (float): longitud de la base.
        altura (float): longitud de la altura.

    Devuelve:
        float: área del rectángulo (base * altura).
    """
    return base * altura

print("Área:", area_rectangulo(5, 3))

# Para ver la documentación:
# help(area_rectangulo)
```

CAPÍTULO 11. PROGRAMACIÓN ORIENTADA A OBJETOS (POO) EN PYTHON

11.1 Conceptos fundamentales

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código alrededor de **objetos**, en lugar de hacerlo solo alrededor de funciones y datos sueltos. Un **objeto** combina **datos** (atributos) y **comportamientos** (métodos) en una misma unidad lógica.

En POO, los conceptos clave son:

- **Clase:**
  Es un "molde" o "plantilla" que describe cómo serán los objetos: qué datos tendrán y qué podrán hacer.
- **Objeto (instancia):**
  Es un "ejemplar" concreto creado a partir de una clase. Si la clase es el plano de una casa, el objeto es la casa construida.
- **Atributos:**
  Son las variables asociadas a un objeto (o a la clase) que almacenan información.
- **Métodos:**
  Son funciones definidas dentro de una clase que describen el comportamiento de los objetos.
- **Encapsulación:**
  Es la idea de agrupar datos y métodos dentro de una clase y controlar el acceso a ellos.
- **Herencia:**
  Permite crear nuevas clases basadas en clases existentes, reutilizando y extendiendo su comportamiento.
- **Polimorfismo:**
  Permite usar una misma interfaz (mismo nombre de método) con comportamientos diferentes según el tipo de objeto.

Ejemplo conceptual sencillo

Imagina que quieres modelar **coches** en un programa:

- Clase: `Coche`
- Atributos: `marca`, `modelo`, `color`, `velocidad_actual`
- Métodos: `acelerar()`, `frenar()`, `pitar()`

Cada coche concreto (tu coche, el de tu vecino, etc.) será un **objeto** de la clase `Coche`.

11.2 Clases y objetos

En Python, una clase se define con la palabra clave `class`.

```
class Coche:
    pass
```

Esta clase no hace nada todavía, pero ya es una clase válida. Para crear un **objeto** (instancia) de esta clase:

```
mi_coche = Coche()
otro_coche = Coche()
```

`mi_coche` y `otro_coche` son dos objetos distintos, aunque de la misma clase.

Añadiendo un constructor (init)

Normalmente, queremos que al crear un objeto se inicialicen algunos datos. Para eso usamos el método especial `__init__`.

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.velocidad = 0
```

- **self:**
  Es una referencia al propio objeto. Se usa para acceder a sus atributos y métodos.
- **marca, modelo, color:**
  Son parámetros que se pasan al crear el objeto.

Crear objetos:

```
coche1 = Coche("Toyota", "Corolla", "Rojo")
coche2 = Coche("Ford", "Focus", "Azul")

print(coche1.marca)   # Toyota
print(coche2.color)   # Azul
```

11.3 Atributos y métodos

Atributos de instancia

Son los que pertenecen a cada objeto individual. En el ejemplo anterior: `marca`, `modelo`, `color`, `velocidad` son atributos de instancia.

```
print(coche1.velocidad)  # 0
coche1.velocidad = 50
print(coche1.velocidad)  # 50
print(coche2.velocidad)  # 0 (no cambia)
```

Cada objeto tiene su propio valor de los atributos.

Métodos de instancia

Son funciones definidas dentro de la clase que operan sobre los atributos del objeto.

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.velocidad = 0

    def acelerar(self, cantidad):
        self.velocidad += cantidad

    def frenar(self, cantidad):
        self.velocidad -= cantidad
        if self.velocidad < 0:
            self.velocidad = 0

    def mostrar_info(self):
        print(f"{self.marca} {self.modelo} ({self.color}) -
Velocidad: {self.velocidad} km/h")
```

Uso:

```
coche = Coche("Seat", "Ibiza", "Blanco")
coche.mostrar_info()      # Seat Ibiza (Blanco) - Velocidad: 0
km/h
coche.acelerar(30)
coche.mostrar_info()      # Velocidad: 30
coche.frenar(50)
coche.mostrar_info()      # Velocidad: 0
```

Atributos de clase

Son compartidos por todas las instancias de la clase.

```
class Coche:
    ruedas = 4  # atributo de clase
```

```
    def __init__(self, marca):
        self.marca = marca  # atributo de instancia

c1 = Coche("Toyota")
c2 = Coche("Ford")

print(c1.ruedas, c2.ruedas)  # 4 4
Coche.ruedas = 6
print(c1.ruedas, c2.ruedas)  # 6 6
```

11.4 Encapsulación

La **encapsulación** consiste en:

- Agrupar datos y métodos relacionados dentro de una clase.
- Controlar el acceso a esos datos (ocultar detalles internos).

En Python no existe un sistema de privacidad tan estricto como en otros lenguajes, pero se usan **convenciones**:

- **Atributo público:** `self.nombre`
- **Atributo "protegido" (convención):** `self._nombre`
- **Atributo "privado" (name mangling):** `self.__nombre`

Ejemplo de encapsulación básica

```
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.__saldo = saldo_inicial  # "privado"

    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad

    def retirar(self, cantidad):
        if 0 < cantidad <= self.__saldo:
            self.__saldo -= cantidad
```

```
        else:
            print("Operación no válida")

    def obtener_saldo(self):
        return self.__saldo
```

Uso:

```
cuenta = CuentaBancaria("Luis", 1000)
cuenta.depositar(500)
print(cuenta.obtener_saldo())  # 1500
cuenta.retirar(2000)           # Operación no válida
print(cuenta.obtener_saldo())  # 1500
```

Si intentas acceder directamente a `cuenta.__saldo`, fallará:

```
# print(cuenta.__saldo)  # AttributeError
```

Python internamente renombra `__saldo` a `_CuentaBancaria__saldo`, pero la idea es que **no lo uses directamente**.

11.5 Herencia

La **herencia** permite crear una nueva clase (clase hija o derivada) a partir de otra (clase base o padre), reutilizando su código y añadiendo o modificando funcionalidades.

```
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hablar(self):
        print("El animal hace un sonido genérico")

class Perro(Animal):
    def hablar(self):
        print(f"{self.nombre} dice: ¡Guau!")

class Gato(Animal):
    def hablar(self):
        print(f"{self.nombre} dice: ¡Miau!")
```

Uso:

```
perro = Perro("Firulais")
gato = Gato("Misu")

perro.hablar()  # Firulais dice: ¡Guau!
gato.hablar()   # Misu dice: ¡Miau!
```

super()

Cuando una clase hija quiere usar el comportamiento de la clase padre, se usa `super()`.

```
class Vehiculo:
    def __init__(self, marca):
        self.marca = marca

class Coche(Vehiculo):
    def __init__(self, marca, modelo):
        super().__init__(marca)  # llama al __init__ de Vehiculo
        self.modelo = modelo
```

11.6 Polimorfismo

El **polimorfismo** permite que diferentes clases implementen un mismo método con comportamientos distintos, y que el código que los usa no tenga que preocuparse del tipo concreto.

```
class Ave:
    def volar(self):
        print("El ave vuela")

class Avion:
    def volar(self):
        print("El avión despega y vuela")

def hacer_volar(objeto):
    objeto.volar()
```

Uso:

```
pajaro = Ave()
```

```
boeing = Avion()


hacer_volar(pajaro)  # El ave vuela
hacer_volar(boeing)  # El avión despega y vuela
```

Aquí, `hacer_volar` no sabe si es un `Ave` o un `Avion`, solo sabe que tiene un método `volar`.

Esto se conoce también como **duck typing**: “si camina como un pato y suena como un pato, entonces es un pato”.

11.7 Métodos especiales

Python tiene muchos **métodos especiales** (también llamados “dunder methods”, por “double underscore”) que permiten integrar nuestras clases con el lenguaje.

Algunos importantes:

- **`__init__(self, ...)`**: Constructor.
- **`__str__(self)`**: Representación “bonita” en forma de cadena, para `print()`.
- **`__repr__(self)`**: Representación oficial, útil para depuración.
- **`__len__(self)`**: Longitud del objeto, para `len(obj)`.
- **`__eq__(self, other)`**: Comparación de igualdad `==`.

Ejemplo con str y repr

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

    def __repr__(self):
        return f"Persona(nombre={self.nombre!r},
edad={self.edad})"
```

Uso:

```
p = Persona("Ana", 30)
print(p)        # usa __str__
print([p])      # la lista usa __repr__ para mostrar el objeto
```

Ejemplo con len

```
class Grupo:
    def __init__(self, nombre):
        self.nombre = nombre
        self.miembros = []

    def agregar(self, persona):
        self.miembros.append(persona)

    def __len__(self):
        return len(self.miembros)
```

Uso:

```
g = Grupo("Amigos")
g.agregar("Luis")
g.agregar("Marta")
print(len(g))  # 2
```

11.8 Buenas prácticas en POO con Python

- **Nombrado claro:**
  Usa nombres de clases en `CamelCase` (`CuentaBancaria`, `CocheDeportivo`) y nombres de métodos/atributos en `snake_case` (`depositar_dinero`, `saldo_actual`).
- **Una clase, una responsabilidad:**
  Cada clase debería tener una responsabilidad clara y no mezclar demasiadas cosas.
- **Evitar duplicación de código:**
  Si varias clases comparten lógica, considera usar herencia o composición.
- **Encapsular detalles internos:**
  No expongas atributos que no deberían modificarse desde fuera. Usa métodos o propiedades.
- **Usar métodos especiales con sentido:**
  Implementa `__str__`, `__repr__`, `__len__`, etc., solo cuando aporten claridad.
- **Documentar:**
  Añade docstrings a clases y métodos para explicar qué hacen.

Ejemplo de clase bien estructurada:

```
class Rectangulo:
```

```
    """
    Representa un rectángulo en 2D.
    """

    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        """Devuelve el área del rectángulo."""
        return self.ancho * self.alto

    def perimetro(self):
        """Devuelve el perímetro del rectángulo."""
        return 2 * (self.ancho + self.alto)

    def __str__(self):
        return f"Rectángulo {self.ancho}x{self.alto}"
```

---

20 EJERCICIOS PROPUESTOS (SIN SOLUCIÓN AQUÍ)

Importante: Las soluciones están al final de la lección. Intenta primero resolverlos tú.

Ejercicio 1:
Crea una clase `Persona` con atributos `nombre` y `edad`. Añade un método `presentarse()` que imprima:
"Hola, me llamo NOMBRE y tengo EDAD años". Crea dos personas y haz que se presenten.

Ejercicio 2:
Crea una clase `Coche` con atributos `marca`, `modelo` y `velocidad` (inicialmente 0). Añade métodos `acelerar(cantidad)` y `frenar(cantidad)` que modifiquen la velocidad sin permitir valores negativos. Muestra la velocidad tras varias operaciones.

Ejercicio 3:
Crea una clase `CuentaBancaria` con atributos `titular` y `saldo`. Implementa métodos `depositar(cantidad)` y `retirar(cantidad)` con validaciones (no permitir retirar más de lo que hay). Añade un método `mostrar_saldo()`.

Ejercicio 4:
Crea una clase `Rectangulo` con atributos `ancho` y `alto`. Añade métodos `area()` y `perimetro()`. Crea varios rectángulos y muestra sus áreas y perímetros.

Ejercicio 5:
Crea una clase `Alumno` con atributos `nombre` y `nota`. Añade un método `esta_aprobado()` que devuelva `True` si la nota es mayor o igual que 5, y `False` en caso contrario.

Ejercicio 6:
Crea una clase `Producto` con atributos `nombre`, `precio` y `stock`. Añade un método `vender(cantidad)` que reduzca el stock si hay suficiente, y otro método `reponer(cantidad)` que aumente el stock.

Ejercicio 7:
Crea una clase `Libro` con atributos `titulo`, `autor` y `paginas`. Implementa el método especial `__str__` para que al imprimir un libro se vea: `"TITULO, de AUTOR (PAGINAS páginas)"`.

Ejercicio 8:
Crea una clase `Punto` en 2D con atributos `x` e `y`. Añade un método `mover(dx, dy)` que desplace el punto. Añade también un método `distancia_origen()` que calcule la distancia al origen `(0,0)`.

Ejercicio 9:
Crea una clase `Animal` con un método `hablar()` que imprima un mensaje genérico. Crea dos clases hijas `Perro` y `Gato` que sobrescriban `hablar()` con sonidos específicos. Crea una función que reciba un animal y llame a `hablar()`.

Ejercicio 10:
Crea una clase `Empleado` con atributos `nombre` y `salario`. Crea una clase hija `Gerente` que añada un atributo `departamento`. Sobrescribe un método `mostrar_info()` para que muestre también el departamento.

Ejercicio 11:
Crea una clase `Figura` con un método `area()` que simplemente lance una excepción `NotImplementedError`. Crea clases hijas `Circulo` y `Rectangulo` que implementen `area()` correctamente. Demuestra el polimorfismo calculando el área de varias figuras en una lista.

Ejercicio 12:
Crea una clase `ListaEnteros` que internamente use una lista de Python para almacenar enteros. Implementa los métodos especiales `__len__` y `__str__` para mostrar la longitud y el contenido de forma amigable.

Ejercicio 13:
Crea una clase `Temperatura` con un atributo `celsius`. Añade una propiedad `fahrenheit` que permita leer y escribir la temperatura en Fahrenheit, convirtiendo internamente a Celsius.

Ejercicio 14:
Crea una clase `Contador` con un atributo “privado” `__valor`. Añade métodos

`incrementar()` y `decrementar()` y un método `obtener_valor()`. Comprueba que no se pueda acceder directamente a `__valor` desde fuera (al menos no de forma directa).

Ejercicio 15:
Crea una clase `Vehiculo` con atributos `marca` y `modelo`. Crea una clase hija `Coche` que añada el atributo `puertas`. Crea otra clase hija `Moto` que añada el atributo `cilindrada`. Crea varios objetos y muestra su información.

Ejercicio 16:
Crea una clase `CarritoCompra` que almacene productos (nombre y precio). Añade métodos para `agregar_producto(nombre, precio)`, `calcular_total()` y `mostrar_productos()`.

Ejercicio 17:
Crea una clase `Agenda` que permita guardar contactos (nombre y teléfono). Implementa métodos para `añadir_contacto`, `buscar_contacto` y `mostrar_todos`.

Ejercicio 18:
Crea una clase `Pila` (stack) que implemente las operaciones `apilar`, `desapilar` y `esta_vacia`. Usa una lista interna para almacenar los elementos.

Ejercicio 19:
Crea una clase `Cancion` con atributos `titulo` y `artista`. Implementa `__str__` para mostrar `"TITULO - ARTISTA"`. Crea una clase `Playlist` que contenga una lista de canciones y métodos para `agregar_cancion` y `mostrar_playlist`.

Ejercicio 20:
Crea una clase `Fraccion` con atributos `numerador` y `denominador`. Implementa un método para simplificar la fracción (dividiendo por el máximo común divisor) y el método especial `__str__` para mostrarla como `"numerador/denominador"`.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
        print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años")

p1 = Persona("Ana", 25)
```

```
p2 = Persona("Luis", 40)

p1.presentarse()
p2.presentarse()
```

Solución Ejercicio 2

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.velocidad = 0

    def acelerar(self, cantidad):
        if cantidad > 0:
            self.velocidad += cantidad

    def frenar(self, cantidad):
        if cantidad > 0:
            self.velocidad -= cantidad
            if self.velocidad < 0:
                self.velocidad = 0

    def mostrar_velocidad(self):
        print(f"Velocidad actual: {self.velocidad} km/h")

c = Coche("Seat", "Ibiza")
c.mostrar_velocidad()
c.acelerar(50)
c.mostrar_velocidad()
c.frenar(30)
c.mostrar_velocidad()
c.frenar(50)
```

```
c.mostrar_velocidad()
```

Solución Ejercicio 3

```
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, cantidad):
        if cantidad > 0:
            self.saldo += cantidad

    def retirar(self, cantidad):
        if 0 < cantidad <= self.saldo:
            self.saldo -= cantidad
        else:
            print("No se puede retirar esa cantidad")

    def mostrar_saldo(self):
        print(f"Saldo de {self.titular}: {self.saldo} €")

cuenta = CuentaBancaria("Luis", 1000)
cuenta.mostrar_saldo()
cuenta.depositar(500)
cuenta.mostrar_saldo()
cuenta.retirar(2000)
cuenta.mostrar_saldo()
```

Solución Ejercicio 4

```
class Rectangulo:
    def __init__(self, ancho, alto):
        self.ancho = ancho
```

```
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

    def perimetro(self):
        return 2 * (self.ancho + self.alto)

r1 = Rectangulo(3, 4)
r2 = Rectangulo(5, 2)

print("Rectángulo 1 - Área:", r1.area(), "Perímetro:",
r1.perimetro())
print("Rectángulo 2 - Área:", r2.area(), "Perímetro:",
r2.perimetro())
```

Solución Ejercicio 5

```
class Alumno:
    def __init__(self, nombre, nota):
        self.nombre = nombre
        self.nota = nota

    def esta_aprobado(self):
        return self.nota >= 5

a1 = Alumno("Ana", 7)
a2 = Alumno("Pedro", 4)

print(a1.nombre, "aprobado:", a1.esta_aprobado())
print(a2.nombre, "aprobado:", a2.esta_aprobado())
```

Solución Ejercicio 6

```
class Producto:
```

```
    def __init__(self, nombre, precio, stock):
        self.nombre = nombre
        self.precio = precio
        self.stock = stock

    def vender(self, cantidad):
        if 0 < cantidad <= self.stock:
            self.stock -= cantidad
        else:
            print("No hay suficiente stock")

    def reponer(self, cantidad):
        if cantidad > 0:
            self.stock += cantidad

p = Producto("Teclado", 20.0, 10)
p.vender(3)
print("Stock tras vender:", p.stock)
p.reponer(5)
print("Stock tras reponer:", p.stock)
```

Solución Ejercicio 7

```
class Libro:
    def __init__(self, titulo, autor, paginas):
        self.titulo = titulo
        self.autor = autor
        self.paginas = paginas

    def __str__(self):
        return f"{self.titulo}, de {self.autor} ({self.paginas}
páginas)"
```

```
l = Libro("El Quijote", "Cervantes", 1200)
print(l)
```

Solución Ejercicio 8

```
import math

class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def mover(self, dx, dy):
        self.x += dx
        self.y += dy

    def distancia_origen(self):
        return math.sqrt(self.x**2 + self.y**2)

p = Punto(3, 4)
print("Distancia al origen:", p.distancia_origen())
p.mover(1, -2)
print("Nueva distancia al origen:", p.distancia_origen())
```

Solución Ejercicio 9

```
class Animal:
    def hablar(self):
        print("El animal hace un sonido")

class Perro(Animal):
    def hablar(self):
        print("Guau")
```

```
class Gato(Animal):
    def hablar(self):
        print("Miau")

def hacer_hablar(animal):
    animal.hablar()

perro = Perro()
gato = Gato()

hacer_hablar(perro)
hacer_hablar(gato)
```

Solución Ejercicio 10

```
class Empleado:
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario

    def mostrar_info(self):
        print(f"Empleado: {self.nombre}, Salario: {self.salario}
€")

class Gerente(Empleado):
    def __init__(self, nombre, salario, departamento):
        super().__init__(nombre, salario)
        self.departamento = departamento

    def mostrar_info(self):
        print(f"Gerente: {self.nombre}, Salario: {self.salario} €,
Depto: {self.departamento}")

e = Empleado("Ana", 2000)
```

```
g = Gerente("Luis", 3000, "Ventas")

e.mostrar_info()
g.mostrar_info()
```

Solución Ejercicio 11

```
import math

class Figura:
    def area(self):
        raise NotImplementedError("Este método debe ser
implementado por la subclase")

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return math.pi * self.radio**2

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

figuras = [Circulo(3), Rectangulo(4, 5), Circulo(1)]

for f in figuras:
    print("Área:", f.area())
```

Solución Ejercicio 12

```
class ListaEnteros:
    def __init__(self):
        self._datos = []

    def agregar(self, valor):
        if isinstance(valor, int):
            self._datos.append(valor)
        else:
            print("Solo se permiten enteros")

    def __len__(self):
        return len(self._datos)

    def __str__(self):
        return f"ListaEnteros({self._datos})"

lista = ListaEnteros()
lista.agregar(1)
lista.agregar(2)
lista.agregar("hola")  # no se agrega
print(lista)
print("Longitud:", len(lista))
```

Solución Ejercicio 13

```
class Temperatura:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
```

```
        return self._celsius

    @celsius.setter
    def celsius(self, valor):
        self._celsius = valor

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, valor):
        self._celsius = (valor - 32) * 5/9

t = Temperatura(0)
print("Celsius:", t.celsius, "Fahrenheit:", t.fahrenheit)
t.fahrenheit = 212
print("Celsius:", t.celsius, "Fahrenheit:", t.fahrenheit)
```

Solución Ejercicio 14

```
class Contador:
    def __init__(self, valor_inicial=0):
        self.__valor = valor_inicial

    def incrementar(self):
        self.__valor += 1

    def decrementar(self):
        self.__valor -= 1

    def obtener_valor(self):
        return self.__valor
```

```
c = Contador()
c.incrementar()
c.incrementar()
c.decrementar()
print("Valor del contador:", c.obtener_valor())

# print(c.__valor)  # Esto daría error
```

Solución Ejercicio 15

```
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar_info(self):
        print(f"{self.marca} {self.modelo}")

class Coche(Vehiculo):
    def __init__(self, marca, modelo, puertas):
        super().__init__(marca, modelo)
        self.puertas = puertas

    def mostrar_info(self):
        print(f"Coche: {self.marca} {self.modelo}, {self.puertas}
puertas")

class Moto(Vehiculo):
    def __init__(self, marca, modelo, cilindrada):
        super().__init__(marca, modelo)
        self.cilindrada = cilindrada
```

```
    def mostrar_info(self):
        print(f"Moto: {self.marca} {self.modelo}, {self.cilindrada} cc")

c = Coche("Seat", "León", 5)
m = Moto("Yamaha", "R3", 300)

c.mostrar_info()
m.mostrar_info()
```

Solución Ejercicio 16

```
class CarritoCompra:
    def __init__(self):
        self.productos = []  # cada producto será una tupla (nombre, precio)

    def agregar_producto(self, nombre, precio):
        self.productos.append((nombre, precio))

    def calcular_total(self):
        return sum(precio for _, precio in self.productos)

    def mostrar_productos(self):
        for nombre, precio in self.productos:
            print(f"{nombre}: {precio} €")

carrito = CarritoCompra()
carrito.agregar_producto("Pan", 1.0)
carrito.agregar_producto("Leche", 1.2)
carrito.mostrar_productos()
print("Total:", carrito.calcular_total(), "€")
```

Solución Ejercicio 17

```
class Agenda:
    def __init__(self):
        self.contactos = {}  # nombre -> teléfono

    def añadir_contacto(self, nombre, telefono):
        self.contactos[nombre] = telefono

    def buscar_contacto(self, nombre):
        if nombre in self.contactos:
            print(f"{nombre}: {self.contactos[nombre]}")
        else:
            print("Contacto no encontrado")

    def mostrar_todos(self):
        for nombre, telefono in self.contactos.items():
            print(f"{nombre}: {telefono}")

agenda = Agenda()
agenda.añadir_contacto("Ana", "123456789")
agenda.añadir_contacto("Luis", "987654321")
agenda.buscar_contacto("Ana")
agenda.mostrar_todos()
```

Solución Ejercicio 18

```
class Pila:
    def __init__(self):
        self._datos = []

    def apilar(self, elemento):
        self._datos.append(elemento)
```

```
    def desapilar(self):
        if self.esta_vacia():
            print("La pila está vacía")
            return None
        return self._datos.pop()

    def esta_vacia(self):
        return len(self._datos) == 0

pila = Pila()
pila.apilar(1)
pila.apilar(2)
pila.apilar(3)
print(pila.desapilar())
print(pila.desapilar())
print(pila.desapilar())
print(pila.desapilar())
```

Solución Ejercicio 19

```
class Cancion:
    def __init__(self, titulo, artista):
        self.titulo = titulo
        self.artista = artista

    def __str__(self):
        return f"{self.titulo} - {self.artista}"

class Playlist:
    def __init__(self, nombre):
        self.nombre = nombre
        self.canciones = []
```

```
    def agregar_cancion(self, cancion):
        self.canciones.append(cancion)

    def mostrar_playlist(self):
        print(f"Playlist: {self.nombre}")
        for c in self.canciones:
            print(c)

c1 = Cancion("Song 1", "Artist A")
c2 = Cancion("Song 2", "Artist B")

pl = Playlist("Favoritas")
pl.agregar_cancion(c1)
pl.agregar_cancion(c2)
pl.mostrar_playlist()
```

Solución Ejercicio 20

```
import math

class Fraccion:
    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador
        self.simplificar()

    def simplificar(self):
        mcd = math.gcd(self.numerador, self.denominador)
        self.numerador //= mcd
        self.denominador //= mcd

    def __str__(self):
        return f"{self.numerador}/{self.denominador}"
```

```
f = Fraccion(8, 12)
print(f)  # 2/3
```

CAPÍTULO 12. TRATAMIENTO DE ARCHIVOS Y FICHEROS EN PYTHON

12.1 Apertura y cierre de archivos

Trabajar con archivos es una de las tareas más habituales en programación: leer datos, guardar resultados, registrar logs, etc. En Python, el flujo básico es:

1. **Abrir el archivo** (con `open()`).
2. **Trabajar con él** (leer o escribir).
3. **Cerrar el archivo** (con `close()` o usando `with`, que lo cierra automáticamente).

La función open() básica

La forma más simple de abrir un archivo es:

```
f = open("datos.txt", "r", encoding="utf-8")
```

- "**datos.txt**": ruta o nombre del archivo.
- "**r**": modo de apertura (lectura).
- **encoding="utf-8**": codificación de texto (muy recomendable especificarla).

Cuando terminas de usar el archivo, debes cerrarlo:

```
f.close()
```

Cerrar el archivo es importante porque:

- Libera recursos del sistema.
- Asegura que los datos escritos se guarden correctamente.
- Evita bloqueos o corrupción de datos.

Ejemplo sencillo de apertura y cierre

```
f = open("saludo.txt", "w", encoding="utf-8")
f.write("Hola, mundo\n")
f.close()
```

Este código:

- Abre (o crea) `saludo.txt` en modo escritura.
- Escribe una línea.
- Cierra el archivo.

12.2 Lectura y escritura de archivos

Una vez abierto el archivo, podemos **leer** o **escribir** según el modo.

Lectura de archivos de texto

Método read(): lee todo el contenido como una sola cadena.

```
f = open("texto.txt", "r", encoding="utf-8")
contenido = f.read()
```

```
f.close()


print(contenido)
```

Método readline(): lee una sola línea cada vez.

```
f = open("texto.txt", "r", encoding="utf-8")
linea1 = f.readline()
linea2 = f.readline()
f.close()

print("Línea 1:", linea1)
print("Línea 2:", linea2)
```

Método readlines(): lee todas las líneas y devuelve una lista de cadenas.

```
f = open("texto.txt", "r", encoding="utf-8")
lineas = f.readlines()
f.close()

for linea in lineas:
    print(linea.strip())
```

Iterar directamente sobre el archivo (forma muy habitual y eficiente):

```
with open("texto.txt", "r", encoding="utf-8") as f:
    for linea in f:
        print(linea.strip())
```

Escritura de archivos de texto

write(): escribe una cadena tal cual.

```
with open("salida.txt", "w", encoding="utf-8") as f:
    f.write("Primera línea\n")
    f.write("Segunda línea\n")
```

writelines(): escribe una lista de cadenas (no añade saltos de línea automáticamente).

```
lineas = ["Línea 1\n", "Línea 2\n", "Línea 3\n"]

with open("lista.txt", "w", encoding="utf-8") as f:
    f.writelines(lineas)
```

12.3 Modos de apertura

El segundo parámetro de `open()` indica el **modo de apertura**. Los más usados:

- "**r**": lectura (error si no existe).
- "**w**": escritura (crea o sobrescribe).
- "**a**": añadir al final (append).
- "**x**": creación exclusiva (error si ya existe).
- "**b**": modo binario (se combina con otros: `"rb"`, `"wb"`, etc.).
- "**+**": lectura y escritura (se combina: `"r+"`, `"w+"`, `"a+"`).

Ejemplos de modos

Modo lectura "**r**"

```
with open("datos.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

Modo escritura "**w**" (sobrescribe el archivo)

```
with open("log.txt", "w", encoding="utf-8") as f:
    f.write("Inicio del log\n")
```

Modo añadir "**a**" (no borra lo anterior, añade al final)

```
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Nueva entrada de log\n")
```

Modo lectura/escritura "**r+**"

```
with open("datos.txt", "r+", encoding="utf-8") as f:
    contenido = f.read()
    f.seek(0)  # volver al inicio
    f.write("Cabecera añadida\n" + contenido)
```

Modo binario "**rb**" / "**wb**" (para imágenes, audio, etc.)

```
# Copiar un archivo binario
with open("imagen_origen.jpg", "rb") as f_in:
    datos = f_in.read()

with open("imagen_copia.jpg", "wb") as f_out:
    f_out.write(datos)
```

12.4 Gestión de rutas

No siempre trabajamos con archivos en el mismo directorio del script. Necesitamos manejar **rutas**:

- **Rutas absolutas:** indican la ubicación completa desde la raíz del sistema.
- **Rutas relativas:** se basan en el directorio actual.

Rutas relativas

```
# Archivo en una carpeta "datos" dentro del directorio actual
with open("datos/usuarios.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

Uso de os.path (forma clásica)

```
import os

directorio = "datos"
nombre_archivo = "usuarios.txt"

ruta = os.path.join(directorio, nombre_archivo)

with open(ruta, "r", encoding="utf-8") as f:
    print(f.read())
```

`os.path.join` construye rutas correctas según el sistema operativo.

Uso de pathlib (forma moderna y muy recomendable)

```
from pathlib import Path

base = Path("datos")
ruta = base / "usuarios.txt"

with open(ruta, "r", encoding="utf-8") as f:
    print(f.read())
```

`Path` permite:

- **Comprobar existencia:**

```
if ruta.exists():
    print("El archivo existe")
else:
```

```
    print("No existe")
```

- **Crear directorios:**

```
carpeta = Path("logs")
carpeta.mkdir(exist_ok=True)
```

12.5 with open(): el gestor de contexto

Usar `open()` y luego `close()` manualmente funciona, pero es fácil olvidarse de cerrar el archivo o que ocurra una excepción antes de `close()`.

La forma **recomendada** es usar `with`, que crea un **gestor de contexto**:

```
with open("archivo.txt", "r", encoding="utf-8") as f:
    contenido = f.read()
    print(contenido)
# Aquí el archivo ya está cerrado automáticamente
```

Ventajas:

- Cierre automático del archivo, incluso si hay errores dentro del bloque.
- Código más limpio y legible.
- Menos riesgo de fugas de recursos.

Ejemplo comparativo

Sin with:

```
f = open("datos.txt", "r", encoding="utf-8")
try:
    for linea in f:
        print(linea.strip())
finally:
    f.close()
```

Con with (equivalente, pero más simple):

```
with open("datos.txt", "r", encoding="utf-8") as f:
    for linea in f:
        print(linea.strip())
```

12.6 Errores comunes con archivos

Trabajar con archivos puede generar errores frecuentes. Algunos de los más típicos:

- **Archivo no encontrado (`FileNotFoundError`)**
  Ocurre cuando intentas abrir un archivo en modo lectura y no existe.

```
try:
    with open("no_existe.txt", "r", encoding="utf-8") as f:
        print(f.read())
except FileNotFoundError:
    print("El archivo no existe")
```

- **Permiso denegado (`PermissionError`)**
  Ocurre cuando no tienes permisos para leer o escribir en el archivo o carpeta.
- **Olvidar cerrar el archivo**
  Si no usas `with`, puedes dejar archivos abiertos, lo que puede bloquearlos o consumir recursos.
- **Sobrescribir archivos por error**
  Abrir en modo `"w"` borra el contenido anterior. Hay que tener cuidado.
- **Problemas de codificación (`UnicodeDecodeError`, `UnicodeEncodeError`)**
  Ocurren cuando la codificación usada no coincide con la del archivo. Por eso es muy recomendable usar siempre `encoding="utf-8"` y ser consistente.

```
try:
    with open("texto.txt", "r", encoding="utf-8") as f:
        print(f.read())
except UnicodeDecodeError:
    print("Error de codificación al leer el archivo")
```

- **Rutas incorrectas**
  Si la ruta no es correcta (carpeta que no existe, error de nombre, etc.), obtendrás `FileNotFoundError`. Conviene comprobar rutas con `Path.exists()` o imprimir la ruta completa para depurar.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Crea un programa que abra un archivo llamado `mensaje.txt` en modo escritura y escriba dentro la frase:
"Este es un mensaje de prueba". Cierra el archivo correctamente.

Ejercicio 2:
Crea un programa que lea el contenido completo de `mensaje.txt` y lo muestre por pantalla.

Ejercicio 3:
Crea un programa que lea un archivo llamado `lineas.txt` línea a línea (usando un bucle `for`) y muestre cada línea numerada (por ejemplo: "Línea 1: ...").

Ejercicio 4:
Crea un programa que abra un archivo `datos.txt` en modo escritura y pida al usuario que introduzca líneas de texto por teclado. Cada línea se escribirá en el archivo. El programa terminará cuando el usuario introduzca una línea vacía.

Ejercicio 5:
Crea un programa que abra un archivo `numeros.txt` en modo lectura, lea todos los números (uno por línea), los convierta a enteros y calcule la suma total. Muestra el resultado.

Ejercicio 6:
Crea un programa que copie el contenido de un archivo de texto `origen.txt` a otro archivo `copia.txt`.

Ejercicio 7:
Crea un programa que abra un archivo `log.txt` en modo añadir (`"a"`) y escriba una línea con el texto "Nueva ejecución del programa". Ejecuta el programa varias veces y comprueba el contenido del archivo.

Ejercicio 8:
Crea un programa que abra un archivo binario (por ejemplo, `imagen.jpg`) en modo lectura binaria (`"rb"`) y lo copie a `imagen_copia.jpg` usando lectura y escritura en binario.

Ejercicio 9:
Crea un programa que use `os.path.join` para construir la ruta de un archivo `config.txt` dentro de una carpeta `config`. Abre el archivo en modo lectura y muestra su contenido (puedes crear el archivo a mano).

Ejercicio 10:
Crea un programa que use `pathlib.Path` para:

- Crear una carpeta `salidas` si no existe.
- Crear dentro un archivo `resultado.txt` y escribir "Proceso completado".

Ejercicio 11:
Crea un programa que abra un archivo `texto_largo.txt` y cuente cuántas líneas tiene, cuántas palabras en total y cuántos caracteres. Muestra un pequeño resumen por pantalla.

Ejercicio 12:
Crea un programa que lea un archivo `usuarios.txt` donde cada línea tiene el formato `nombre;edad`. El programa debe leer cada línea, separar nombre y edad, y mostrar un mensaje del tipo: "Nombre: X, Edad: Y".

Ejercicio 13:
Crea un programa que abra un archivo `datos.txt` en modo `"r+"`. Lee su contenido, vuelve al inicio con `seek(0)` y escribe al principio la línea "CABECERA DEL ARCHIVO".

Ejercicio 14:
Crea un programa que intente abrir un archivo `inexistente.txt` en modo lectura. Captura la excepción `FileNotFoundError` y muestra un mensaje amigable en lugar de que el programa se detenga con error.

Ejercicio 15:
Crea un programa que pida al usuario una ruta de archivo por teclado. Si el archivo existe, muestra su tamaño en bytes. Si no existe, muestra un mensaje indicándolo. Usa `pathlib.Path`.

Ejercicio 16:
Crea un programa que lea un archivo `texto.txt` y cree otro archivo `texto_mayusculas.txt` donde se guarde el mismo contenido pero todo en mayúsculas.

Ejercicio 17:
Crea un programa que lea un archivo `lista_compra.txt` (una línea por producto) y genere un archivo `lista_compra_numerada.txt` donde cada línea esté numerada (por ejemplo: "1. Pan").

Ejercicio 18:
Crea un programa que abra un archivo `entrada.txt` y cuente cuántas veces aparece una palabra concreta (por ejemplo, "Python"). La palabra se pedirá al usuario por teclado.

Ejercicio 19:
Crea un programa que lea un archivo `temperaturas.txt` donde cada línea contiene una temperatura en grados Celsius. El programa debe crear un archivo `temperaturas_fahrenheit.txt` con las temperaturas convertidas a Fahrenheit.

Ejercicio 20:
Crea un programa que lea un archivo `alumnos.txt` donde cada línea tiene el formato `nombre;nota`. El programa debe crear un archivo `aprobados.txt` con solo los alumnos cuya nota sea mayor o igual que 5.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
with open("mensaje.txt", "w", encoding="utf-8") as f:
    f.write("Este es un mensaje de prueba\n")
```

Solución Ejercicio 2

```
with open("mensaje.txt", "r", encoding="utf-8") as f:
    contenido = f.read()

print(contenido)
```

Solución Ejercicio 3

```
with open("lineas.txt", "r", encoding="utf-8") as f:
    for numero, linea in enumerate(f, start=1):
        print(f"Línea {numero}: {linea.strip()}")
```

Solución Ejercicio 4

```
with open("datos.txt", "w", encoding="utf-8") as f:
    while True:
        texto = input("Introduce una línea (vacía para terminar):
")
        if texto == "":
            break
        f.write(texto + "\n")
```

Solución Ejercicio 5

```
suma = 0

with open("numeros.txt", "r", encoding="utf-8") as f:
    for linea in f:
        linea = linea.strip()
        if linea:
            numero = int(linea)
            suma += numero

print("La suma total es:", suma)
```

Solución Ejercicio 6

```
with open("origen.txt", "r", encoding="utf-8") as f_origen:
    contenido = f_origen.read()

with open("copia.txt", "w", encoding="utf-8") as f_destino:
    f_destino.write(contenido)
```

Solución Ejercicio 7

```
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Nueva ejecución del programa\n")
```

Solución Ejercicio 8

```
with open("imagen.jpg", "rb") as f_in:
    datos = f_in.read()

with open("imagen_copia.jpg", "wb") as f_out:
    f_out.write(datos)
```

Solución Ejercicio 9

```
import os

carpeta = "config"
nombre_archivo = "config.txt"

ruta = os.path.join(carpeta, nombre_archivo)

with open(ruta, "r", encoding="utf-8") as f:
    contenido = f.read()

print(contenido)
```

Solución Ejercicio 10

```
from pathlib import Path

carpeta = Path("salidas")
carpeta.mkdir(exist_ok=True)

ruta_archivo = carpeta / "resultado.txt"
```

```
with open(ruta_archivo, "w", encoding="utf-8") as f:
    f.write("Proceso completado\n")
```

Solución Ejercicio 11

```
num_lineas = 0
num_palabras = 0
num_caracteres = 0

with open("texto_largo.txt", "r", encoding="utf-8") as f:
    for linea in f:
        num_lineas += 1
        num_caracteres += len(linea)
        palabras = linea.split()
        num_palabras += len(palabras)

print("Resumen del archivo:")
print("Líneas:", num_lineas)
print("Palabras:", num_palabras)
print("Caracteres:", num_caracteres)
```

Solución Ejercicio 12

```
with open("usuarios.txt", "r", encoding="utf-8") as f:
    for linea in f:
        linea = linea.strip()
        if not linea:
            continue
        nombre, edad = linea.split(";")
        print(f"Nombre: {nombre}, Edad: {edad}")
```

Solución Ejercicio 13

```
with open("datos.txt", "r+", encoding="utf-8") as f:
```

```
    contenido = f.read()
    f.seek(0)
    f.write("CABECERA DEL ARCHIVO\n")
    f.write(contenido)
```

Solución Ejercicio 14

```
try:
    with open("inexistente.txt", "r", encoding="utf-8") as f:
        print(f.read())
except FileNotFoundError:
    print("El archivo 'inexistente.txt' no existe. Comprueba el nombre o la ruta.")
```

Solución Ejercicio 15

```
from pathlib import Path

ruta_texto = input("Introduce la ruta del archivo: ")
ruta = Path(ruta_texto)

if ruta.exists() and ruta.is_file():
    tamaño = ruta.stat().st_size
    print("El archivo existe. Tamaño en bytes:", tamaño)
else:
    print("El archivo no existe.")
```

Solución Ejercicio 16

```
with open("texto.txt", "r", encoding="utf-8") as f_entrada:
    contenido = f_entrada.read()

with open("texto_mayusculas.txt", "w", encoding="utf-8") as f_salida:
    f_salida.write(contenido.upper())
```

Solución Ejercicio 17

```
with open("lista_compra.txt", "r", encoding="utf-8") as f_entrada:
    lineas = f_entrada.readlines()

with open("lista_compra_numerada.txt", "w", encoding="utf-8") as
f_salida:
    for i, linea in enumerate(lineas, start=1):
        producto = linea.strip()
        if producto:
            f_salida.write(f"{i}. {producto}\n")
```

Solución Ejercicio 18

```
palabra = input("Introduce la palabra a buscar: ")

contador = 0

with open("entrada.txt", "r", encoding="utf-8") as f:
    for linea in f:
        palabras = linea.split()
        for p in palabras:
            if p == palabra:
                contador += 1

print(f"La palabra '{palabra}' aparece {contador} veces.")
```

(Si quisieras una búsqueda más flexible, podrías pasar todo a minúsculas y quitar signos de puntuación.)

Solución Ejercicio 19

```
def celsius_a_fahrenheit(c):
    return c * 9/5 + 32

with open("temperaturas.txt", "r", encoding="utf-8") as f_entrada,
\
```

```
     open("temperaturas_fahrenheit.txt", "w", encoding="utf-8") as
f_salida:

    for linea in f_entrada:
        linea = linea.strip()
        if not linea:
            continue
        celsius = float(linea)
        fahrenheit = celsius_a_fahrenheit(celsius)
        f_salida.write(f"{fahrenheit}\n")
```

Solución Ejercicio 20

```
with open("alumnos.txt", "r", encoding="utf-8") as f_entrada, \
     open("aprobados.txt", "w", encoding="utf-8") as f_salida:

    for linea in f_entrada:
        linea = linea.strip()
        if not linea:
            continue
        nombre, nota_str = linea.split(";")
        nota = float(nota_str)
        if nota >= 5:
            f_salida.write(f"{nombre};{nota}\n")
```

CAPÍTULO 13. FICHEROS CSV EN PYTHON

13.0 Introducción a los ficheros CSV

Un **fichero CSV** (*Comma-Separated Values*, valores separados por comas) es un tipo de archivo de texto muy usado para almacenar datos en forma de **tabla**: filas y columnas.

- Cada **línea** del archivo representa una **fila**.
- Dentro de cada línea, los **campos** (columnas) se separan por un **delimitador** (normalmente una coma `,`, pero también puede ser `;`, `\t`, etc.).

Ejemplo de CSV sencillo (con `;` como delimitador):

```
nombre;edad;ciudad
Ana;30;Madrid
Luis;25;Zaragoza
Marta;40;Valencia
```

Python incluye el módulo estándar **`csv`**, que facilita mucho la lectura y escritura de este tipo de archivos, evitando que tengamos que partir cadenas manualmente.

En este capítulo veremos:

- **13.1** Lectura con `csv.reader`
- **13.2** Escritura con `csv.writer`
- **13.3** Lectura y escritura con `DictReader` y `DictWriter`
- **13.4** Delimitadores y dialectos
- **13.5** Errores típicos y cómo evitarlos

13.1 Lectura con csv.reader

El objeto **`csv.reader`** permite leer un fichero CSV y obtener cada fila como una **lista de cadenas**.

Pasos básicos para leer un CSV

1. Importar el módulo `csv`.
2. Abrir el archivo con `open()` (modo lectura, texto, con codificación).
3. Crear un objeto `csv.reader` a partir del archivo.
4. Recorrer el reader en un bucle para procesar cada fila.

Ejemplo básico de lectura

Supongamos un archivo `personas.csv` con contenido:

```
nombre,edad,ciudad
Ana,30,Madrid
Luis,25,Zaragoza
```

```
Marta,40,Valencia
```

Código:

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f)  # por defecto delimitador = ","
    for fila in lector:
        print(fila)
```

Salida:

```
['nombre', 'edad', 'ciudad']
['Ana', '30', 'Madrid']
['Luis', '25', 'Zaragoza']
['Marta', '40', 'Valencia']
```

Observa que:

- Cada `fila` es una **lista de cadenas**.
- Los valores numéricos (como `30`) se leen como texto y, si los necesitas como números, tendrás que convertirlos (`int`, `float`, etc.).

Saltar la cabecera

Muchas veces el CSV tiene una **fila de cabecera** con los nombres de las columnas. Si quieres saltarla:

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f)
    cabecera = next(lector)  # lee la primera fila
    print("Cabecera:", cabecera)

    for fila in lector:
        print("Datos:", fila)
```

Conversión de tipos

Ejemplo: convertir la edad a entero:

```
import csv
```

```
with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f)
    next(lector)  # saltar cabecera

    for fila in lector:
        nombre = fila[0]
        edad = int(fila[1])
        ciudad = fila[2]
        print(f"{nombre} tiene {edad} años y vive en {ciudad}")
```

13.2 Escritura con csv.writer

El objeto **`csv.writer`** permite escribir filas en un fichero CSV a partir de listas (o iterables) de valores.

Pasos básicos para escribir un CSV

1. Importar `csv`.
2. Abrir el archivo en modo escritura (`"w"` o `"a"`) con `open()`.
3. Crear un objeto `csv.writer`.
4. Usar `writerow()` para escribir una fila o `writerows()` para varias.

Ejemplo básico de escritura

```
import csv

with open("salida.csv", "w", encoding="utf-8", newline="") as f:
    escritor = csv.writer(f)
    # Escribir cabecera
    escritor.writerow(["nombre", "edad", "ciudad"])
    # Escribir filas de datos
    escritor.writerow(["Ana", 30, "Madrid"])
    escritor.writerow(["Luis", 25, "Zaragoza"])
    escritor.writerow(["Marta", 40, "Valencia"])
```

Observa:

- `newline=""` es recomendable en Windows para evitar líneas en blanco extra.
- `csv.writer` se encarga de convertir los valores a texto y separarlos con el delimitador.

Escribir varias filas a la vez

```
import csv

filas = [
    ["nombre", "edad", "ciudad"],
    ["Ana", 30, "Madrid"],
    ["Luis", 25, "Zaragoza"],
    ["Marta", 40, "Valencia"],
]

with open("salida2.csv", "w", encoding="utf-8", newline="") as f:
    escritor = csv.writer(f)
    escritor.writerows(filas)
```

13.3 DictReader y DictWriter

A veces es más cómodo trabajar con **diccionarios** en lugar de listas, usando los nombres de las columnas como claves. Para eso están **`DictReader`** y **`DictWriter`**.

Lectura con DictReader

`csv.DictReader` lee cada fila y la devuelve como un **diccionario** donde:

- Las **claves** son los nombres de las columnas (tomados de la cabecera).
- Los **valores** son las cadenas leídas.

Ejemplo con `personas.csv`:

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)  # usa la primera fila como cabecera
    for fila in lector:
        print(fila)
```

Salida aproximada:

```
{'nombre': 'Ana', 'edad': '30', 'ciudad': 'Madrid'}
{'nombre': 'Luis', 'edad': '25', 'ciudad': 'Zaragoza'}
{'nombre': 'Marta', 'edad': '40', 'ciudad': 'Valencia'}
```

Acceso por nombre de columna:

```
import csv


with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        nombre = fila["nombre"]
        edad = int(fila["edad"])
        ciudad = fila["ciudad"]
        print(f"{nombre} ({edad}) - {ciudad}")
```

Escritura con DictWriter

`csv.DictWriter` permite escribir filas a partir de diccionarios. Necesita saber qué **campos** (columnas) tendrá el CSV.

```
import csv


campos = ["nombre", "edad", "ciudad"]


with open("personas_dict.csv", "w", encoding="utf-8", newline="")
as f:
    escritor = csv.DictWriter(f, fieldnames=campos)


    # Escribir cabecera
    escritor.writeheader()


    # Escribir filas
    escritor.writerow({"nombre": "Ana", "edad": 30, "ciudad":
"Madrid"})
    escritor.writerow({"nombre": "Luis", "edad": 25, "ciudad":
"Zaragoza"})
    escritor.writerow({"nombre": "Marta", "edad": 40, "ciudad":
"Valencia"})
```

Ventajas de `DictWriter`:

- No dependes del orden de las columnas al construir la lista.

- Puedes construir diccionarios con claridad y el writer se encarga de colocarlos en el orden correcto.

13.4 Delimitadores y dialectos

No todos los CSV usan la coma `,` como separador. En muchos países (como España) es muy común usar el **punto y coma `;`** porque la coma se usa como separador decimal.

El módulo `csv` permite configurar:

- **Delimitador** (`delimiter`)
- **Carácter de comillas** (`quotechar`)
- **Cómo se manejan las comillas** (`quoting`)
- Otros detalles (escape, salto de línea, etc.)

Cambiar el delimitador

Supongamos un archivo `personas_puntoycoma.csv`:

```
nombre;edad;ciudad
Ana;30;Madrid
Luis;25;Zaragoza
Marta;40;Valencia
```

Lectura con `delimiter=";"`:

```
import csv

with open("personas_puntoycoma.csv", "r", encoding="utf-8",
newline="") as f:
    lector = csv.reader(f, delimiter=";")
    for fila in lector:
        print(fila)
```

Escritura con `delimiter=";"`:

```
import csv

with open("salida_puntoycoma.csv", "w", encoding="utf-8",
newline="") as f:
    escritor = csv.writer(f, delimiter=";")
    escritor.writerow(["nombre", "edad", "ciudad"])
    escritor.writerow(["Ana", 30, "Madrid"])
    escritor.writerow(["Luis", 25, "Zaragoza"])
```

Dialectos

Un **dialecto** es una configuración predefinida de parámetros de CSV. Puedes:

- Usar dialectos predefinidos (como `"excel"`).
- Registrar tus propios dialectos.

Ejemplo de registro de dialecto:

```
import csv


csv.register_dialect(
    "puntoycoma",
    delimiter=";",
    quotechar='"',
    quoting=csv.QUOTE_MINIMAL
)


with open("personas_puntoycoma.csv", "r", encoding="utf-8",
newline="") as f:
    lector = csv.reader(f, dialect="puntoycoma")
    for fila in lector:
        print(fila)
```

También puedes usar el dialecto `"excel"` (por defecto) o `"excel-tab"` (tabulador).

Comillas y campos con separadores internos

Si un campo contiene el delimitador o saltos de línea, se suele encerrar entre comillas. Ejemplo:

```
nombre,comentario
Ana,"Me gusta Python, es genial"
Luis,"Vive en ""Zaragoza"" y le encanta"
```

El módulo `csv` maneja esto automáticamente si se usan los parámetros adecuados (`quotechar`, `quoting`).

13.5 Errores típicos con CSV

Al trabajar con CSV es fácil cometer errores. Algunos de los más comunes:

- **Olvidar `newline=""` al abrir el archivo en Windows**
  Puede provocar líneas en blanco extra al escribir.

- **Usar el delimitador incorrecto**
  Si el archivo usa `;` y tú lees con `,`, obtendrás filas con un solo campo (toda la línea).
- **No especificar la codificación correcta**
  Si el archivo está en UTF-8 pero lo abres sin `encoding` o con otra codificación, puedes tener errores de lectura o caracteres raros.
- **Suponer que todos los campos son válidos**
  Puede haber líneas vacías, campos faltantes, valores no numéricos donde esperas números, etc. Conviene validar y usar `try/except`.
- **Confundir el orden de las columnas**
  Si trabajas con `csv.reader` (listas), es fácil equivocarse de índice. `DictReader` ayuda a evitar esto.

Ejemplo de manejo de errores básicos:

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        try:
            edad = int(fila["edad"])
        except ValueError:
            print("Edad no válida en fila:", fila)
            continue

        print(f"{fila['nombre']} tiene {edad} años")
```

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Crea un archivo `personas.csv` con las columnas `nombre,edad,ciudad` y al menos 3 filas de datos. Escribe un programa en Python que lo lea con `csv.reader` y muestre cada fila por pantalla.

Ejercicio 2:
Modifica el programa anterior para que salte la cabecera y muestre solo los datos, indicando: `"Nombre: X, Edad: Y, Ciudad: Z"`.

Ejercicio 3:
Crea un programa que lea `personas.csv` con `csv.DictReader` y muestre solo los nombres de las personas cuya edad sea mayor o igual que 30.

Ejercicio 4:
Crea un programa que lea `personas.csv` con `csv.DictReader`, convierta la edad a entero y calcule la edad media de todas las personas.

Ejercicio 5:
Crea un programa que genere un archivo `productos.csv` con las columnas `nombre,precio,stock` usando `csv.writer`. Escribe al menos 4 productos.

Ejercicio 6:
Crea un programa que lea `productos.csv` con `csv.DictReader` y muestre el valor total del inventario (suma de `precio * stock` de todos los productos).

Ejercicio 7:
Crea un programa que lea `personas.csv` y genere un nuevo archivo `personas_mayores.csv` con solo las personas cuya edad sea mayor o igual que 18. Usa `DictReader` y `DictWriter`.

Ejercicio 8:
Crea un archivo `alumnos.csv` con columnas `nombre;nota` usando `;` como delimitador. Escribe un programa que lo lea con `csv.reader` configurando `delimiter=";"` y muestre qué alumnos están aprobados (nota >= 5).

Ejercicio 9:
Repite el ejercicio anterior pero usando `csv.DictReader` con `delimiter=";"`.

Ejercicio 10:
Crea un programa que lea un archivo `ventas.csv` con columnas `producto,cantidad,precio_unitario` y calcule el total de ventas (suma de `cantidad * precio_unitario`). Usa `DictReader`.

Ejercicio 11:
Crea un programa que lea `personas.csv` y genere un archivo `personas_ordenadas.csv` con las mismas columnas pero ordenado por edad de menor a mayor. Puedes leer todo en una lista de diccionarios, ordenarla y luego escribirla con `DictWriter`.

Ejercicio 12:
Crea un programa que lea un archivo `empleados.csv` con columnas `nombre,salario` y aumente el salario de todos un 10%, guardando el resultado en `empleados_actualizados.csv`.

Ejercicio 13:
Crea un dialecto llamado `"puntoycoma"` que use `;` como delimitador. Úsalo para leer un archivo `datos_puntoycoma.csv` y mostrar su contenido.

Ejercicio 14:
Crea un programa que lea un archivo `datos_malos.csv` donde algunas filas tienen una edad no numérica. Usa `DictReader` y maneja el error con `try/except`, mostrando un mensaje cuando la edad no sea válida y saltando esa fila.

Ejercicio 15:
Crea un programa que lea un archivo `inventario.csv` con columnas `producto,categoria,stock`. Pide al usuario una categoría por teclado y muestra solo los productos de esa categoría.

Ejercicio 16:
Crea un programa que lea un archivo `temperaturas.csv` con columnas `ciudad,temperatura_c`. Convierte las temperaturas a Fahrenheit y genera un archivo `temperaturas_f.csv` con columnas `ciudad,temperatura_f`.

Ejercicio 17:
Crea un programa que lea un archivo `notas.csv` con columnas `nombre,nota1,nota2,nota3`. Calcula la media de cada alumno y genera un archivo `notas_media.csv` con columnas `nombre,media`.

Ejercicio 18:
Crea un programa que lea un archivo `clientes.csv` con columnas `id,nombre,email`. Comprueba que todos los emails contienen el carácter `@`. Si alguno no lo contiene, muestra un mensaje indicando el `id` del cliente con email inválido.

Ejercicio 19:
Crea un programa que lea un archivo `libros.csv` con columnas `titulo,autor,anio`. Filtra los libros publicados a partir del año 2000 y genera `libros_modernos.csv` con solo esos libros.

Ejercicio 20:
Crea un programa que lea un archivo `transacciones.csv` con columnas `fecha,descripcion,importe`. Calcula el saldo acumulado (sumando los importes) y muestra por pantalla cada transacción junto con el saldo tras aplicarla.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
import csv


with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f)
    for fila in lector:
        print(fila)
```

Solución Ejercicio 2

```
import csv
```

```
with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f)
    cabecera = next(lector)  # saltar cabecera

    for fila in lector:
        nombre = fila[0]
        edad = fila[1]
        ciudad = fila[2]
        print(f"Nombre: {nombre}, Edad: {edad}, Ciudad: {ciudad}")
```

Solución Ejercicio 3

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        edad = int(fila["edad"])
        if edad >= 30:
            print(fila["nombre"])
```

Solución Ejercicio 4

```
import csv

suma_edades = 0
contador = 0

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        edad = int(fila["edad"])
        suma_edades += edad
        contador += 1
```

```
if contador > 0:
    media = suma_edades / contador
    print("Edad media:", media)
else:
    print("No hay datos")
```

Solución Ejercicio 5

```
import csv

productos = [
    ["nombre", "precio", "stock"],
    ["Teclado", 20.5, 10],
    ["Ratón", 15.0, 25],
    ["Monitor", 150.0, 5],
    ["USB", 8.0, 50],
]

with open("productos.csv", "w", encoding="utf-8", newline="") as
f:
    escritor = csv.writer(f)
    escritor.writerows(productos)
```

Solución Ejercicio 6

```
import csv

total = 0.0

with open("productos.csv", "r", encoding="utf-8", newline="") as
f:
    lector = csv.DictReader(f)
    for fila in lector:
        precio = float(fila["precio"])
```

```
        stock = int(fila["stock"])
        total += precio * stock

print("Valor total del inventario:", total)
```

Solución Ejercicio 7

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as
f_entrada, \
     open("personas_mayores.csv", "w", encoding="utf-8",
newline="") as f_salida:

    lector = csv.DictReader(f_entrada)
    campos = lector.fieldnames  # mismas columnas

    escritor = csv.DictWriter(f_salida, fieldnames=campos)
    escritor.writeheader()

    for fila in lector:
        edad = int(fila["edad"])
        if edad >= 18:
            escritor.writerow(fila)
```

Solución Ejercicio 8

```
import csv

with open("alumnos.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.reader(f, delimiter=";")
    for fila in lector:
        nombre = fila[0]
        nota = float(fila[1])
        if nota >= 5:
```

```
            print(f"{nombre} está aprobado con nota {nota}")
```

Solución Ejercicio 9

```
import csv

with open("alumnos.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f, delimiter=";")
    for fila in lector:
        nombre = fila["nombre"]
        nota = float(fila["nota"])
        if nota >= 5:
            print(f"{nombre} está aprobado con nota {nota}")
```

(Asegúrate de que la cabecera del CSV sea `nombre;nota`.)

Solución Ejercicio 10

```
import csv

total_ventas = 0.0

with open("ventas.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        cantidad = float(fila["cantidad"])
        precio_unitario = float(fila["precio_unitario"])
        total_ventas += cantidad * precio_unitario

print("Total de ventas:", total_ventas)
```

Solución Ejercicio 11

```
import csv

with open("personas.csv", "r", encoding="utf-8", newline="") as f:
```

```
    lector = csv.DictReader(f)
    filas = list(lector)

# Ordenar por edad (convertida a int)
filas.sort(key=lambda fila: int(fila["edad"]))

campos = ["nombre", "edad", "ciudad"]

with open("personas_ordenadas.csv", "w", encoding="utf-8",
newline="") as f_salida:
    escritor = csv.DictWriter(f_salida, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(filas)
```

Solución Ejercicio 12

```
import csv

with open("empleados.csv", "r", encoding="utf-8", newline="") as
f_entrada:
    lector = csv.DictReader(f_entrada)
    filas = list(lector)

for fila in filas:
    salario = float(fila["salario"])
    salario_nuevo = salario * 1.10
    fila["salario"] = salario_nuevo

campos = ["nombre", "salario"]

with open("empleados_actualizados.csv", "w", encoding="utf-8",
newline="") as f_salida:
    escritor = csv.DictWriter(f_salida, fieldnames=campos)
    escritor.writeheader()
```

```
    escritor.writerows(filas)
```

Solución Ejercicio 13

```
import csv

csv.register_dialect(
    "puntoycoma",
    delimiter=";",
    quotechar='"',
    quoting=csv.QUOTE_MINIMAL
)

with open("datos_puntoycoma.csv", "r", encoding="utf-8",
newline="") as f:
    lector = csv.reader(f, dialect="puntoycoma")
    for fila in lector:
        print(fila)
```

Solución Ejercicio 14

```
import csv

with open("datos_malos.csv", "r", encoding="utf-8", newline="") as
f:
    lector = csv.DictReader(f)
    for fila in lector:
        try:
            edad = int(fila["edad"])
        except ValueError:
            print("Edad no válida en fila:", fila)
            continue

        print(f"Fila correcta: {fila['nombre']} ({edad})")
```

Solución Ejercicio 15

```
import csv


categoria_buscada = input("Introduce la categoría: ")


with open("inventario.csv", "r", encoding="utf-8", newline="") as
f:
    lector = csv.DictReader(f)
    for fila in lector:
        if fila["categoria"] == categoria_buscada:
            print(f"Producto: {fila['producto']}, Stock:
{fila['stock']}")
```

Solución Ejercicio 16

```
import csv


def c_a_f(c):
    return c * 9/5 + 32


with open("temperaturas.csv", "r", encoding="utf-8", newline="")
as f_entrada, \
     open("temperaturas_f.csv", "w", encoding="utf-8", newline="")
as f_salida:


    lector = csv.DictReader(f_entrada)
    campos_salida = ["ciudad", "temperatura_f"]
    escritor = csv.DictWriter(f_salida, fieldnames=campos_salida)
    escritor.writeheader()


    for fila in lector:
        ciudad = fila["ciudad"]
        temp_c = float(fila["temperatura_c"])
        temp_f = c_a_f(temp_c)
```

```
        escritor.writerow({"ciudad": ciudad, "temperatura_f":
temp_f})
```

Solución Ejercicio 17

```
import csv


with open("notas.csv", "r", encoding="utf-8", newline="") as
f_entrada, \
     open("notas_media.csv", "w", encoding="utf-8", newline="") as
f_salida:


    lector = csv.DictReader(f_entrada)
    campos_salida = ["nombre", "media"]
    escritor = csv.DictWriter(f_salida, fieldnames=campos_salida)
    escritor.writeheader()


    for fila in lector:
        nombre = fila["nombre"]
        n1 = float(fila["nota1"])
        n2 = float(fila["nota2"])
        n3 = float(fila["nota3"])
        media = (n1 + n2 + n3) / 3
        escritor.writerow({"nombre": nombre, "media": media})
```

Solución Ejercicio 18

```
import csv


with open("clientes.csv", "r", encoding="utf-8", newline="") as f:
    lector = csv.DictReader(f)
    for fila in lector:
        email = fila["email"]
        if "@" not in email:
            print(f"Email inválido para id {fila['id']}: {email}")
```

Solución Ejercicio 19

```
import csv


with open("libros.csv", "r", encoding="utf-8", newline="") as
f_entrada, \
     open("libros_modernos.csv", "w", encoding="utf-8",
newline="") as f_salida:


    lector = csv.DictReader(f_entrada)
    campos = lector.fieldnames
    escritor = csv.DictWriter(f_salida, fieldnames=campos)
    escritor.writeheader()


    for fila in lector:
        anio = int(fila["anio"])
        if anio >= 2000:
            escritor.writerow(fila)
```

Solución Ejercicio 20

```
import csv


saldo = 0.0


with open("transacciones.csv", "r", encoding="utf-8", newline="")
as f:
    lector = csv.DictReader(f)
    for fila in lector:
        importe = float(fila["importe"])
        saldo += importe
        print(f"Fecha: {fila['fecha']}, Desc:
{fila['descripcion']}, "
              f"Importe: {importe}, Saldo acumulado: {saldo}")
```

CAPÍTULO 14. FICHEROS JSON EN PYTHON

14.0 Introducción a JSON

JSON (*JavaScript Object Notation*) es un formato de texto muy usado para intercambiar datos entre programas, servicios web, APIs, etc. Es:

- **Humano-legible** (se puede leer fácilmente).
- **Basado en texto** (no binario).
- **Estructurado** (objetos, listas, valores).
- **Independiente de lenguaje** (no es exclusivo de Python).

En Python, el módulo estándar **`json`** permite:

- Leer JSON desde **ficheros** o **cadenas de texto** y convertirlo a estructuras Python (diccionarios, listas, etc.).
- Convertir estructuras Python a JSON y guardarlas en **ficheros** o **cadenas de texto**.

En este capítulo veremos:

- **14.1** `json.load` y `json.dump` (trabajo con ficheros).
- **14.2** `json.loads` y `json.dumps` (trabajo con cadenas).
- **14.3** Conversión entre diccionarios/listas y JSON.
- **14.4** Errores comunes y cómo evitarlos.

14.1 json.load y json.dump (ficheros JSON)

Cuando trabajamos con **ficheros JSON**, las funciones principales son:

- **`json.load(archivo)`**: lee JSON desde un fichero abierto y devuelve el objeto Python correspondiente.
- **`json.dump(objeto, archivo, ...)`**: escribe un objeto Python en un fichero en formato JSON.

Leer JSON desde un fichero con json.load

Supongamos un fichero `persona.json` con este contenido:

```
{
    "nombre": "Ana",
    "edad": 30,
    "ciudad": "Madrid",
    "aficiones": ["leer", "viajar", "programar"]
}
```

Código para leerlo:

```
import json
```

```
with open("persona.json", "r", encoding="utf-8") as f:
    datos = json.load(f)

print(datos)
print(type(datos))
```

Salida aproximada:

```
{'nombre': 'Ana', 'edad': 30, 'ciudad': 'Madrid', 'aficiones': ['leer', 'viajar', 'programar']}
<class 'dict'>
```

Observa:

- El JSON se ha convertido en un **diccionario de Python**.
- Las listas JSON se convierten en **listas de Python**.
- Los números JSON se convierten en `int` o `float`.
- `true/false` JSON se convierten en `True/False` de Python.
- `null` JSON se convierte en `None`.

Escribir JSON en un fichero con json.dump

Supongamos que tenemos un diccionario en Python:

```
import json

persona = {
    "nombre": "Luis",
    "edad": 25,
    "ciudad": "Zaragoza",
    "aficiones": ["música", "deporte"]
}

with open("persona2.json", "w", encoding="utf-8") as f:
    json.dump(persona, f)
```

Esto creará un fichero `persona2.json` con contenido JSON (en una sola línea por defecto).

Formato “bonito” (indentación)

Para que el JSON sea más legible, podemos usar el parámetro **`indent`**:

```
with open("persona2_pretty.json", "w", encoding="utf-8") as f:
```

```
    json.dump(persona, f, indent=4, ensure_ascii=False)
```

- **`indent=4`**: usa 4 espacios de sangría por nivel.
- **`ensure_ascii=False`**: permite caracteres no ASCII (acentos, ñ, etc.) sin escaparlos.

Ejemplo de salida:

```
{
    "nombre": "Luis",
    "edad": 25,
    "ciudad": "Zaragoza",
    "aficiones": [
        "música",
        "deporte"
    ]
}
```

14.2 json.loads y json.dumps (cadenas JSON)

Cuando los datos JSON vienen de una **cadena de texto** (por ejemplo, de una API web), usamos:

- **`json.loads(cadena)`**: convierte una cadena JSON en un objeto Python.
- **`json.dumps(objeto, ...)`**: convierte un objeto Python en una cadena JSON.

json.loads: de cadena JSON a objeto Python

```
import json

texto_json = '''
{
    "titulo": "Curso de Python",
    "nivel": "básico",
    "duracion": 40,
    "temas": ["variables", "funciones", "ficheros", "JSON"]
}
'''

datos = json.loads(texto_json)
```

```
print(datos)
print(type(datos))
print(datos["titulo"])
print(datos["temas"][2])
```

- `datos` es un diccionario.
- Podemos acceder a sus claves como en cualquier diccionario.

json.dumps: de objeto Python a cadena JSON

```
import json

curso = {
    "titulo": "Curso de Python",
    "nivel": "intermedio",
    "duracion": 60,
    "temas": ["POO", "ficheros", "módulos", "JSON"]
}

cadena = json.dumps(curso)
print(cadena)
print(type(cadena))
```

Salida (una sola línea):

```
{"titulo": "Curso de Python", "nivel": "intermedio", "duracion": 60, "temas": ["POO", "ficheros", "módulos", "JSON"]}
```

json.dumps con formato legible

```
cadena_pretty = json.dumps(curso, indent=4, ensure_ascii=False)
print(cadena_pretty)
```

Salida:

```
{
    "titulo": "Curso de Python",
    "nivel": "intermedio",
    "duracion": 60,
    "temas": [
        "POO",
        "ficheros",
```

```
        "módulos",
        "JSON"
    ]
}
```

14.3 Diccionarios ↔ JSON (y listas)

La relación entre JSON y Python es muy directa:

- **Objeto JSON** (`{ ... }`) ↔ **diccionario** (`dict`)
- **Array JSON** (`[ ... ]`) ↔ **lista** (`list`)
- **String JSON ↔ str**
- **Number JSON ↔ int/float**
- **true/false JSON ↔ True/False**
- **null JSON ↔ None**

De diccionario Python a JSON (fichero)

```
import json

config = {
    "version": 1,
    "debug": True,
    "servidor": {
        "host": "localhost",
        "puerto": 8080
    },
    "usuarios": ["admin", "invitado"]
}

with open("config.json", "w", encoding="utf-8") as f:
    json.dump(config, f, indent=4, ensure_ascii=False)
```

De JSON (fichero) a diccionario Python

```
import json

with open("config.json", "r", encoding="utf-8") as f:
    config = json.load(f)
```

```
print(config["version"])
print(config["servidor"]["host"])
print(config["usuarios"])
```

Listas de diccionarios

Es muy común tener un JSON que es una **lista de objetos**. Ejemplo `personas.json`:

```
[
    {
        "nombre": "Ana",
        "edad": 30
    },
    {
        "nombre": "Luis",
        "edad": 25
    },
    {
        "nombre": "Marta",
        "edad": 40
    }
]
```

Leerlo:

```
import json

with open("personas.json", "r", encoding="utf-8") as f:
    personas = json.load(f)

for p in personas:
    print(p["nombre"], "tiene", p["edad"], "años")
```

Crear y guardar una lista de diccionarios:

```
import json

personas = [
```

```
    {"nombre": "Ana", "edad": 30},
    {"nombre": "Luis", "edad": 25},
    {"nombre": "Marta", "edad": 40},
]

with open("personas2.json", "w", encoding="utf-8") as f:
    json.dump(personas, f, indent=4, ensure_ascii=False)
```

Conversión manual diccionario ↔ JSON (cadenas)

```
import json

persona = {"nombre": "Carlos", "edad": 35}

# A cadena JSON
cadena = json.dumps(persona, ensure_ascii=False)
print("JSON:", cadena)

# De cadena JSON a diccionario
persona2 = json.loads(cadena)
print("Diccionario:", persona2)
```

14.4 Errores comunes con JSON

Trabajar con JSON es sencillo, pero hay errores típicos que conviene conocer.

1. JSON mal formado (json.JSONDecodeError)

Si el contenido no es JSON válido, `json.load` o `json.loads` lanzan `json.JSONDecodeError`.

Ejemplo de JSON incorrecto:

```
{ nombre: "Ana", edad: 30 }   // Falta comillas en las claves, y comentario no permitido
```

JSON **no permite**:

- Comentarios (`//` o `/* ... */`).
- Claves sin comillas.
- Comas finales extra.

Manejo del error:

```
import json

texto = "{ nombre: 'Ana' }"  # JSON incorrecto

try:
    datos = json.loads(texto)
except json.JSONDecodeError as e:
    print("Error al decodificar JSON:", e)
```

2. Confundir JSON con diccionario Python

JSON es **texto**, un diccionario es un **objeto Python**.

- `json.loads` espera una **cadena JSON**, no un diccionario.
- `json.dumps` espera un **objeto Python**, no una cadena JSON.

Ejemplo incorrecto:

```
import json

d = {"a": 1}
# json.loads(d)  # ERROR: loads espera una cadena, no un dict
```

Correcto:

```
cadena = json.dumps(d)
d2 = json.loads(cadena)
```

3. Problemas de codificación (UTF-8)

Es recomendable abrir siempre los ficheros JSON con `encoding="utf-8"` y usar `ensure_ascii=False` si quieres mantener caracteres especiales.

```
with open("datos.json", "r", encoding="utf-8") as f:
    datos = json.load(f)
```

4. Tipos no serializables

`json.dump` / `json.dumps` solo pueden convertir tipos básicos (dict, list, str, int, float, bool, None).

Si intentas serializar un objeto propio (por ejemplo, una instancia de una clase), obtendrás un error `TypeError`.

Ejemplo:

```
import json
from datetime import datetime
```

```
datos = {
    "ahora": datetime.now()  # datetime no es serializable por defecto
}

# json.dumps(datos)  # TypeError
```

Solución sencilla: convertir a cadena:

```
datos = {
    "ahora": datetime.now().isoformat()
}
json.dumps(datos)
```

5. Sobrescribir ficheros por error

Al usar `"w"` en `open`, si el fichero existe se sobrescribe. Hay que tener cuidado y, si es necesario, hacer copias de seguridad o comprobar antes.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Crea un diccionario con tus datos (nombre, edad, ciudad, lista de aficiones) y guárdalo en un fichero `mi_perfil.json` usando `json.dump`.

Ejercicio 2:
Escribe un programa que lea el fichero `mi_perfil.json` con `json.load` y muestre los datos por pantalla de forma legible.

Ejercicio 3:
Crea una lista de diccionarios llamada `alumnos`, donde cada diccionario tenga `nombre` y `nota`. Guarda la lista en `alumnos.json`.

Ejercicio 4:
Escribe un programa que lea `alumnos.json` y muestre solo los nombres de los alumnos aprobados (nota >= 5).

Ejercicio 5:
Crea una cadena JSON (no fichero) que represente un producto con campos `nombre`, `precio` y `en_stock`. Usa `json.loads` para convertirla en diccionario y muestra sus campos.

Ejercicio 6:
Crea un diccionario `config` con campos `version`, `debug` (True/False) y `opciones` (lista de cadenas). Convierte el diccionario a una cadena JSON con `json.dumps` y muéstrala con indentación.

Ejercicio 7:
Crea un fichero `config_app.json` con una estructura de configuración (por ejemplo, idioma, tema, tamaño_ventana). Escribe un programa que lo lea, cambie el valor de `tema` y lo vuelva a guardar.

Ejercicio 8:
Crea un fichero `personas.json` que contenga una lista de personas (nombre, edad). Escribe un programa que calcule la edad media.

Ejercicio 9:
Crea un fichero `tareas.json` con una lista de tareas, cada una con `titulo` y `completada` (True/False). Escribe un programa que muestre solo las tareas no completadas.

Ejercicio 10:
Crea una cadena JSON que contenga un objeto con una lista de números. Usa `json.loads` para convertirla a Python y calcula la suma de los números.

Ejercicio 11:
Escribe un programa que pida al usuario su nombre y edad por teclado, cree un diccionario con esos datos y lo guarde en `usuario.json`.

Ejercicio 12:
Crea un programa que lea `usuario.json` y muestre un mensaje del tipo: `"Hola, NOMBRE, tienes EDAD años"`.

Ejercicio 13:
Crea un fichero `libros.json` con una lista de libros (titulo, autor, anio). Escribe un programa que filtre los libros publicados a partir del año 2000 y los muestre.

Ejercicio 14:
Crea un fichero `inventario.json` con una lista de productos (nombre, precio, stock). Escribe un programa que calcule el valor total del inventario (precio * stock).

Ejercicio 15:
Escribe una cadena JSON incorrecta (por ejemplo, sin comillas en las claves) y usa `json.loads` dentro de un `try/except` para capturar `json.JSONDecodeError` y mostrar un mensaje de error.

Ejercicio 16:
Crea un diccionario que contenga una fecha como objeto `datetime`. Intenta serializarlo con `json.dumps` y observa el error. Luego modifica el código para convertir la fecha a cadena ISO antes de serializar.

Ejercicio 17:
Crea un fichero `config_servidor.json` con campos `host`, `puerto` y `ssl` (True/False). Escribe un programa que lo lea y muestre un mensaje del tipo: `"Conectando a HOST:PUERTO con SSL: True/False"`.

Ejercicio 18:
Crea un fichero `notas.json` con una lista de alumnos (nombre, lista de notas). Escribe un programa que calcule la media de cada alumno y la muestre.

Ejercicio 19:
Crea un fichero `agenda.json` con una lista de contactos (nombre, telefono, email). Escribe un programa que pida un nombre por teclado y muestre los datos del contacto si existe.

Ejercicio 20:
Crea un fichero `transacciones.json` con una lista de transacciones (fecha, descripcion, importe). Escribe un programa que calcule el saldo total (suma de importes) y lo muestre.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
import json

perfil = {
    "nombre": "Luis",
    "edad": 25,
    "ciudad": "Zaragoza",
    "aficiones": ["programar", "música", "cine"]
}

with open("mi_perfil.json", "w", encoding="utf-8") as f:
    json.dump(perfil, f, indent=4, ensure_ascii=False)
```

Solución Ejercicio 2

```
import json

with open("mi_perfil.json", "r", encoding="utf-8") as f:
    perfil = json.load(f)

print("Nombre:", perfil["nombre"])
print("Edad:", perfil["edad"])
print("Ciudad:", perfil["ciudad"])
print("Aficiones:", ", ".join(perfil["aficiones"]))
```

Solución Ejercicio 3

```
import json

alumnos = [
    {"nombre": "Ana", "nota": 7.5},
    {"nombre": "Luis", "nota": 4.0},
    {"nombre": "Marta", "nota": 9.0},
]

with open("alumnos.json", "w", encoding="utf-8") as f:
    json.dump(alumnos, f, indent=4, ensure_ascii=False)
```

Solución Ejercicio 4

```
import json

with open("alumnos.json", "r", encoding="utf-8") as f:
    alumnos = json.load(f)

print("Alumnos aprobados:")
for a in alumnos:
    if a["nota"] >= 5:
        print(a["nombre"], "-", a["nota"])
```

Solución Ejercicio 5

```
import json

cadena_json = '''
{
    "nombre": "Teclado",
    "precio": 25.5,
    "en_stock": true
}
```

```
'''

producto = json.loads(cadena_json)

print("Nombre:", producto["nombre"])
print("Precio:", producto["precio"])
print("En stock:", producto["en_stock"])
```

Solución Ejercicio 6

```
import json

config = {
    "version": 1,
    "debug": True,
    "opciones": ["log", "verbose", "cache"]
}

cadena = json.dumps(config, indent=4, ensure_ascii=False)
print(cadena)
```

Solución Ejercicio 7

```
import json

# Supongamos que config_app.json ya existe con algo como:
# {"idioma": "es", "tema": "claro", "tamaño_ventana": [800, 600]}

with open("config_app.json", "r", encoding="utf-8") as f:
    config = json.load(f)

config["tema"] = "oscuro"

with open("config_app.json", "w", encoding="utf-8") as f:
```

```
    json.dump(config, f, indent=4, ensure_ascii=False)
```

Solución Ejercicio 8

```
import json

with open("personas.json", "r", encoding="utf-8") as f:
    personas = json.load(f)

suma_edades = 0
contador = 0

for p in personas:
    suma_edades += p["edad"]
    contador += 1

if contador > 0:
    media = suma_edades / contador
    print("Edad media:", media)
else:
    print("No hay personas en el fichero")
```

Solución Ejercicio 9

```
import json

with open("tareas.json", "r", encoding="utf-8") as f:
    tareas = json.load(f)

print("Tareas no completadas:")
for t in tareas:
    if not t["completada"]:
        print("-", t["titulo"])
```

Solución Ejercicio 10

```
import json

cadena = '''
{
    "numeros": [1, 2, 3, 4, 5]
}
'''

datos = json.loads(cadena)
numeros = datos["numeros"]
suma = sum(numeros)
print("Suma de números:", suma)
```

Solución Ejercicio 11

```
import json

nombre = input("Introduce tu nombre: ")
edad = int(input("Introduce tu edad: "))

usuario = {
    "nombre": nombre,
    "edad": edad
}

with open("usuario.json", "w", encoding="utf-8") as f:
    json.dump(usuario, f, indent=4, ensure_ascii=False)
```

Solución Ejercicio 12

```
import json

with open("usuario.json", "r", encoding="utf-8") as f:
```

```
    usuario = json.load(f)

print(f"Hola, {usuario['nombre']}, tienes {usuario['edad']} años")
```

Solución Ejercicio 13

```
import json

with open("libros.json", "r", encoding="utf-8") as f:
    libros = json.load(f)

print("Libros publicados a partir del año 2000:")
for libro in libros:
    if libro["anio"] >= 2000:
        print(f"{libro['titulo']} ({libro['anio']}) - {libro['autor']}")
```

Solución Ejercicio 14

```
import json

with open("inventario.json", "r", encoding="utf-8") as f:
    productos = json.load(f)

total = 0.0
for p in productos:
    total += p["precio"] * p["stock"]

print("Valor total del inventario:", total)
```

Solución Ejercicio 15

```
import json

cadena_incorrecta = "{ nombre: 'Ana', edad: 30 }"  # JSON mal formado
```

```
try:
    datos = json.loads(cadena_incorrecta)
except json.JSONDecodeError as e:
    print("Error al decodificar JSON:", e)
```

Solución Ejercicio 16

```
import json
from datetime import datetime

# Versión que falla
datos = {
    "evento": "inicio",
    "fecha": datetime.now()
}

try:
    json.dumps(datos)
except TypeError as e:
    print("Error de serialización:", e)

# Versión corregida
datos_corregidos = {
    "evento": "inicio",
    "fecha": datetime.now().isoformat()
}

cadena = json.dumps(datos_corregidos, indent=4,
ensure_ascii=False)
print(cadena)
```

Solución Ejercicio 17

```
import json

with open("config_servidor.json", "r", encoding="utf-8") as f:
    config = json.load(f)

host = config["host"]
puerto = config["puerto"]
ssl = config["ssl"]

print(f"Conectando a {host}:{puerto} con SSL: {ssl}")
```

Solución Ejercicio 18

```
import json

with open("notas.json", "r", encoding="utf-8") as f:
    alumnos = json.load(f)

for a in alumnos:
    nombre = a["nombre"]
    notas = a["notas"]  # lista de notas
    if len(notas) > 0:
        media = sum(notas) / len(notas)
    else:
        media = 0
    print(f"{nombre}: media = {media}")
```

(Asegúrate de que cada alumno tenga un campo `"notas"` que sea una lista de números.)

Solución Ejercicio 19

```
import json

with open("agenda.json", "r", encoding="utf-8") as f:
```

```
    contactos = json.load(f)

nombre_buscar = input("Introduce el nombre del contacto: ")

encontrado = False
for c in contactos:
    if c["nombre"].lower() == nombre_buscar.lower():
        print("Nombre:", c["nombre"])
        print("Teléfono:", c["telefono"])
        print("Email:", c["email"])
        encontrado = True
        break

if not encontrado:
    print("Contacto no encontrado")
```

Solución Ejercicio 20

```
import json

with open("transacciones.json", "r", encoding="utf-8") as f:
    transacciones = json.load(f)

saldo = 0.0

for t in transacciones:
    importe = t["importe"]
    saldo += importe
    print(f"Fecha: {t['fecha']}, Desc: {t['descripcion']}, "
          f"Importe: {importe}, Saldo acumulado: {saldo}")

print("Saldo final:", saldo)
```

CAPÍTULO 15. TRATAMIENTO DE ERRORES Y EXCEPCIONES EN PYTHON

15.0 Introducción al tratamiento de errores

En cualquier programa real, **los errores son inevitables**: entradas incorrectas, ficheros que no existen, divisiones entre cero, problemas de red, etc. Si no los gestionamos, el programa se detiene de forma brusca mostrando un *traceback* poco amigable.

Python ofrece un sistema muy potente basado en **excepciones** para:

- **Detectar** errores en tiempo de ejecución.
- **Interrumpir** el flujo normal del programa cuando ocurre algo inesperado.
- **Capturar y manejar** esos errores de forma controlada.
- **Recuperarse** del error o, al menos, mostrar un mensaje claro al usuario.

En este capítulo veremos:

- **15.1** `try–except`: captura básica de excepciones.
- **15.2** `else` y `finally`: bloques adicionales para controlar mejor el flujo.
- **15.3** Excepciones personalizadas: crear nuestras propias clases de error.
- **15.4** Lanzamiento de excepciones con `raise`.
- **15.5** Errores frecuentes y buenas prácticas.

15.1 Bloques try–except

El bloque **`try–except`** es la base del manejo de excepciones en Python.

Estructura básica

```
try:
    # Código que puede producir un error
    ...
except:
    # Código que se ejecuta si ocurre un error
    ...
```

Funcionamiento:

- **`try`**: se ejecuta el código dentro del bloque.
- Si **no ocurre ningún error**, se salta el `except`.
- Si **ocurre una excepción**, se interrumpe el bloque `try` y se ejecuta el bloque `except`.

Ejemplo simple: división entre cero

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
```

```
    resultado = x / y
    print("Resultado:", resultado)
except:
    print("Ha ocurrido un error al realizar la división.")
```

Problema: este `except` captura **cualquier** excepción (por ejemplo, error al convertir a entero, división entre cero, etc.). Es mejor ser más específico.

Capturar excepciones específicas

Podemos indicar el **tipo de excepción** que queremos capturar:

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
    resultado = x / y
    print("Resultado:", resultado)
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
```

Ahora solo se captura el error de división entre cero. Si el usuario introduce texto no numérico, el programa seguirá fallando con `ValueError`.

Capturar varias excepciones

Podemos capturar varias excepciones en un mismo `except`:

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
    resultado = x / y
    print("Resultado:", resultado)
except (ZeroDivisionError, ValueError):
    print("Error: división entre cero o valor no numérico.")
```

Capturar la excepción como objeto

Podemos obtener información adicional sobre la excepción:

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
    resultado = x / y
```

```
    print("Resultado:", resultado)
except ZeroDivisionError as e:
    print("Error de división:", e)
except ValueError as e:
    print("Error de conversión:", e)
```

Un `except` general al final

A veces se usa un `except Exception` al final para capturar cualquier error no previsto:

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
    resultado = x / y
    print("Resultado:", resultado)
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
except ValueError:
    print("Debes introducir números enteros.")
except Exception as e:
    print("Ha ocurrido un error inesperado:", e)
```

15.2 Bloques else y finally

Además de `try` y `except`, Python permite usar **`else`** y **`finally`** para controlar mejor el flujo.

Bloque else

La estructura completa es:

```
try:
    # Código que puede lanzar excepción
    ...
except TipoDeError:
    # Se ejecuta si ocurre esa excepción
    ...
else:
    # Se ejecuta solo si NO hubo ninguna excepción en el try
    ...
```

Uso típico:

- `try`: código principal.
- `except`: manejo de errores.
- `else`: código que solo debe ejecutarse si todo fue bien.

Ejemplo:

```
try:
    numero = int(input("Introduce un número entero: "))
except ValueError:
    print("No has introducido un número entero válido.")
else:
    print("Has introducido el número:", numero)
    print("Aquí puedo usar 'numero' con seguridad.")
```

Bloque finally

La estructura con `finally` es:

```
try:
    ...
except TipoDeError:
    ...
finally:
    # Se ejecuta SIEMPRE, haya o no error
    ...
```

Uso típico: liberar recursos, cerrar ficheros, conexiones, etc.

Ejemplo con fichero:

```
f = None
try:
    f = open("datos.txt", "r", encoding="utf-8")
    contenido = f.read()
    print(contenido)
except FileNotFoundError:
    print("El archivo no existe.")
finally:
    if f is not None:
```

```
        f.close()
        print("Archivo cerrado (si estaba abierto).")
```

Combinando try–except–else–finally

Se pueden combinar todos:

```
try:
    x = int(input("Introduce un número: "))
    y = int(input("Introduce otro número: "))
    resultado = x / y
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
except ValueError:
    print("Debes introducir números enteros.")
else:
    print("La división se ha realizado correctamente. Resultado:", resultado)
finally:
    print("Fin del cálculo (con o sin errores).")
```

15.3 Excepciones personalizadas

Python trae muchas excepciones predefinidas (`ValueError`, `TypeError`, `FileNotFoundError`, etc.), pero a veces queremos definir **nuestros propios tipos de error** para representar situaciones específicas de nuestro programa.

Crear una excepción personalizada

Una excepción personalizada es simplemente una **clase** que hereda de `Exception` (o de alguna subclase).

```
class MiError(Exception):
    """Excepción base para errores específicos de mi aplicación."""
    pass
```

Podemos añadir información extra:

```
class EdadInvalidaError(Exception):
    def __init__(self, edad, mensaje="La edad no es válida"):
        self.edad = edad
```

```
        self.mensaje = mensaje
        super().__init__(f"{mensaje}: {edad}")
```

Usar la excepción personalizada

```
def comprobar_edad(edad):
    if edad < 0 or edad > 120:
        raise EdadInvalidaError(edad)
    print("Edad correcta:", edad)


try:
    comprobar_edad(150)
except EdadInvalidaError as e:
    print("Se ha producido un error de edad:", e)
```

Ventajas:

- El código es más **expresivo**.
- Podemos capturar específicamente nuestros errores:

```
try:
    comprobar_edad(-5)
except EdadInvalidaError:
    print("La edad introducida no está en el rango permitido.")
```

15.4 Lanzamiento de excepciones (raise)

Hasta ahora hemos visto cómo **capturar** excepciones. Pero también podemos **lanzarlas** explícitamente con la palabra clave **`raise`**.

Lanzar una excepción estándar

```
def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("El divisor no puede ser cero.")
    return a / b


try:
    print(dividir(10, 0))
except ZeroDivisionError as e:
    print("Error al dividir:", e)
```

Lanzar una excepción personalizada

```
class SaldoInsuficienteError(Exception):
    pass

def retirar_saldo(saldo, cantidad):
    if cantidad > saldo:
        raise SaldoInsuficienteError("No hay saldo suficiente.")
    return saldo - cantidad

try:
    nuevo_saldo = retirar_saldo(100, 200)
except SaldoInsuficienteError as e:
    print("Operación no permitida:", e)
```

Re-lanzar una excepción

A veces capturamos una excepción para hacer algo (por ejemplo, loguearla) y luego la volvemos a lanzar:

```
def procesar():
    try:
        x = int("abc")
    except ValueError as e:
        print("Error al convertir:", e)
        raise  # vuelve a lanzar la misma excepción

try:
    procesar()
except ValueError:
    print("La excepción ha llegado hasta aquí.")
```

15.5 Errores frecuentes y buenas prácticas

Error 1: usar except sin especificar tipo

```
try:
    ...
```

```
except:
    ...
```

Esto captura **todo**, incluso errores de teclado, interrupciones, etc. Es mejor usar:

```
except Exception:
    ...
```

o, mejor aún, **excepciones concretas**.

Error 2: abusar de las excepciones para lógica normal

No es buena idea usar excepciones para controlar el flujo normal del programa cuando se puede usar una condición.

Mal enfoque:

```
try:
    valor = diccionario["clave"]
except KeyError:
    valor = 0
```

Mejor:

```
valor = diccionario.get("clave", 0)
```

Error 3: no mostrar información útil

Si capturas una excepción y solo pones `print("Error")`, no sabrás qué ha pasado.

Mejor:

```
except Exception as e:
    print("Error:", e)
```

O incluso registrar el *traceback* con el módulo `logging` o `traceback`.

Error 4: no limpiar recursos

Si abres ficheros, conexiones, etc., y no usas `finally` o context managers (`with`), puedes dejar recursos abiertos.

Mejor usar `with` o `try-finally`.

Error 5: no validar entradas antes de usarlas

Aunque las excepciones ayudan, es buena práctica **validar** datos antes de operar con ellos.

Ejemplo:

```
texto = input("Introduce un número: ")
if texto.isdigit():
    numero = int(texto)
```

```
else:
    print("No es un número válido.")
```

Se puede combinar validación + excepciones para robustez máxima.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Escribe un programa que pida dos números al usuario y muestre su división. Usa `try-except` para capturar `ZeroDivisionError` y `ValueError`, mostrando mensajes claros.

Ejercicio 2:
Modifica el ejercicio anterior para que, si no hay errores, muestre un mensaje adicional usando un bloque `else`.

Ejercicio 3:
Escribe una función `leer_entero()` que pida un número por teclado y lo devuelva como entero. Si el usuario introduce algo incorrecto, captura `ValueError` y vuelve a pedir el número hasta que sea válido.

Ejercicio 4:
Escribe un programa que abra un fichero `datos.txt` en modo lectura. Captura `FileNotFoundError` y muestra un mensaje amigable si el archivo no existe.

Ejercicio 5:
Crea una función `abrir_fichero(nombre)` que intente abrir un fichero y devuelva su contenido. Usa `try-except-finally` para asegurarte de que el fichero se cierra siempre.

Ejercicio 6:
Define una excepción personalizada llamada `EdadInvalidaError`. Escribe una función `validar_edad(edad)` que lance esa excepción si la edad es menor que 0 o mayor que 120.

Ejercicio 7:
Escribe un programa que pida una edad al usuario, use `validar_edad` y capture `EdadInvalidaError`, mostrando un mensaje adecuado.

Ejercicio 8:
Crea una función `dividir(a, b)` que lance `ValueError` si `b` es 0 con un mensaje `"El divisor no puede ser cero"`. Captura esa excepción al llamar a la función.

Ejercicio 9:
Escribe una función `buscar_elemento(lista, indice)` que intente devolver `lista[indice]`. Captura `IndexError` y devuelve `None` en ese caso.

Ejercicio 10:
Crea una función `convertir_a_entero(cadena)` que intente convertir una cadena a entero. Si falla, lanza una excepción personalizada `ConversionEnteroError`.

Ejercicio 11:
Escribe un programa que lea números de un fichero `numeros.txt` (uno por línea) y calcule su

suma. Captura `ValueError` si alguna línea no es un número válido y muestra un aviso, pero continúa con el resto.

Ejercicio 12:
Crea una función `abrir_config(nombre)` que lea un fichero JSON de configuración. Captura `FileNotFoundError` y `json.JSONDecodeError`, mostrando mensajes distintos para cada caso.

Ejercicio 13:
Escribe un programa que pida al usuario un índice y muestre el carácter correspondiente de una cadena fija. Captura `IndexError` y muestra un mensaje si el índice es incorrecto.

Ejercicio 14:
Crea una función `operacion_peligrosa()` que dentro haga una conversión a entero y una división. Captura las excepciones dentro de la función, muestra un mensaje y luego vuelve a lanzar la excepción.

Ejercicio 15:
Escribe un programa que llame a `operacion_peligrosa()` dentro de un `try-except` y capture la excepción relanzada, mostrando un mensaje final.

Ejercicio 16:
Define una excepción personalizada `SaldoInsuficienteError`. Crea una función `retirar(saldo, cantidad)` que lance esa excepción si `cantidad > saldo`.

Ejercicio 17:
Escribe un programa que use `retirar` y capture `SaldoInsuficienteError`, mostrando un mensaje `"Operación no permitida: saldo insuficiente"`.

Ejercicio 18:
Crea una función `leer_linea_segura(nombre_fichero)` que intente leer la primera línea de un fichero. Usa `try-except-else-finally` para:

- Abrir el fichero.
- Leer la línea.
- Mostrar un mensaje si todo fue bien (`else`).
- Cerrar el fichero siempre (`finally`).

Ejercicio 19:
Escribe un programa que pida un número al usuario y lance manualmente `ValueError` si el número es negativo, con el mensaje `"El número no puede ser negativo"`. Captura esa excepción y muestra el mensaje.

Ejercicio 20:
Crea una pequeña "calculadora" que pida una operación (`+`, `-`, `*`, `/`) y dos números. Usa `try-except` para capturar errores de conversión y división entre cero, y muestra mensajes claros.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
try:
    x = float(input("Introduce el primer número: "))
    y = float(input("Introduce el segundo número: "))
    resultado = x / y
    print("Resultado de la división:", resultado)
except ZeroDivisionError:
    print("Error: no se puede dividir entre cero.")
except ValueError:
    print("Error: debes introducir valores numéricos.")
```

Solución Ejercicio 2

```
try:
    x = float(input("Introduce el primer número: "))
    y = float(input("Introduce el segundo número: "))
    resultado = x / y
except ZeroDivisionError:
    print("Error: no se puede dividir entre cero.")
except ValueError:
    print("Error: debes introducir valores numéricos.")
else:
    print("La división se ha realizado correctamente. Resultado:",
resultado)
```

Solución Ejercicio 3

```
def leer_entero():
    while True:
        try:
            valor = int(input("Introduce un número entero: "))
            return valor
        except ValueError:
```

```
            print("Eso no es un entero válido. Inténtalo de
nuevo.")


n = leer_entero()
print("Has introducido:", n)
```

Solución Ejercicio 4

```
try:
    with open("datos.txt", "r", encoding="utf-8") as f:
        contenido = f.read()
        print(contenido)
except FileNotFoundError:
    print("El archivo 'datos.txt' no existe. Comprueba el nombre o
la ruta.")
```

Solución Ejercicio 5

```
def abrir_fichero(nombre):
    f = None
    try:
        f = open(nombre, "r", encoding="utf-8")
        contenido = f.read()
        return contenido
    except FileNotFoundError:
        print(f"El archivo '{nombre}' no existe.")
        return None
    finally:
        if f is not None:
            f.close()
            # print("Archivo cerrado.")  # opcional

texto = abrir_fichero("datos.txt")
if texto is not None:
    print("Contenido del archivo:")
```

```
    print(texto)
```

Solución Ejercicio 6

```
class EdadInvalidaError(Exception):
    def __init__(self, edad, mensaje="Edad fuera de rango"):
        self.edad = edad
        super().__init__(f"{mensaje}: {edad}")

def validar_edad(edad):
    if edad < 0 or edad > 120:
        raise EdadInvalidaError(edad)
```

Solución Ejercicio 7

```
# Se asume que EdadInvalidaError y validar_edad ya están definidos

try:
    edad = int(input("Introduce tu edad: "))
    validar_edad(edad)
except ValueError:
    print("Debes introducir un número entero para la edad.")
except EdadInvalidaError as e:
    print("Error de edad:", e)
else:
    print("Edad válida:", edad)
```

Solución Ejercicio 8

```
def dividir(a, b):
    if b == 0:
        raise ValueError("El divisor no puede ser cero.")
    return a / b

try:
```

```
    x = float(input("Introduce el numerador: "))
    y = float(input("Introduce el denominador: "))
    r = dividir(x, y)
except ValueError as e:
    print("Error:", e)
else:
    print("Resultado:", r)
```

Solución Ejercicio 9

```
def buscar_elemento(lista, indice):
    try:
        return lista[indice]
    except IndexError:
        return None

datos = [10, 20, 30]
print(buscar_elemento(datos, 1))   # 20
print(buscar_elemento(datos, 5))   # None
```

Solución Ejercicio 10

```
class ConversionEnteroError(Exception):
    pass

def convertir_a_entero(cadena):
    try:
        return int(cadena)
    except ValueError:
        raise ConversionEnteroError(f"No se pudo convertir
'{cadena}' a entero.")

try:
    valor = convertir_a_entero("abc")
```

```
except ConversionEnteroError as e:
    print("Error de conversión:", e)
```

Solución Ejercicio 11

```
suma = 0

try:
    with open("numeros.txt", "r", encoding="utf-8") as f:
        for linea in f:
            linea = linea.strip()
            if not linea:
                continue
            try:
                n = float(linea)
                suma += n
            except ValueError:
                print(f"Línea no numérica ignorada: '{linea}'")
except FileNotFoundError:
    print("El archivo 'numeros.txt' no existe.")
else:
    print("Suma total:", suma)
```

Solución Ejercicio 12

```
import json

def abrir_config(nombre):
    try:
        with open(nombre, "r", encoding="utf-8") as f:
            datos = json.load(f)
            return datos
    except FileNotFoundError:
```

```
        print(f"El archivo de configuración '{nombre}' no
existe.")
    except json.JSONDecodeError as e:
        print(f"El archivo '{nombre}' no contiene JSON válido:",
e)
    return None

config = abrir_config("config.json")
if config is not None:
    print("Configuración cargada:", config)
```

Solución Ejercicio 13

```
cadena = "Hola, mundo"

try:
    indice = int(input("Introduce un índice: "))
    caracter = cadena[indice]
except ValueError:
    print("Debes introducir un número entero.")
except IndexError:
    print("Índice fuera de rango para la cadena.")
else:
    print(f"El carácter en la posición {indice} es '{caracter}'")
```

Solución Ejercicio 14

```
def operacion_peligrosa():
    try:
        x = int(input("Introduce un número entero: "))
        y = int(input("Introduce otro número entero: "))
        r = x / y
        print("Resultado dentro de operacion_peligrosa:", r)
    except Exception as e:
        print("Error dentro de operacion_peligrosa:", e)
```

```
        raise  # relanzar la misma excepción
```

Solución Ejercicio 15

```
try:
    operacion_peligrosa()
except Exception as e:
    print("La excepción relanzada ha sido capturada en el programa principal:", e)
```

Solución Ejercicio 16

```
class SaldoInsuficienteError(Exception):
    pass


def retirar(saldo, cantidad):
    if cantidad > saldo:
        raise SaldoInsuficienteError("Saldo insuficiente para retirar esa cantidad.")
    return saldo - cantidad
```

Solución Ejercicio 17

```
# Se asume que SaldoInsuficienteError y retirar ya están definidos


saldo_actual = 100.0


try:
    cantidad = float(input("Cantidad a retirar: "))
    saldo_actual = retirar(saldo_actual, cantidad)
except ValueError:
    print("Debes introducir una cantidad numérica.")
except SaldoInsuficienteError as e:
    print("Operación no permitida:", e)
else:
    print("Retirada correcta. Saldo restante:", saldo_actual)
```

Solución Ejercicio 18

```
def leer_linea_segura(nombre_fichero):
    f = None
    try:
        f = open(nombre_fichero, "r", encoding="utf-8")
        linea = f.readline()
    except FileNotFoundError:
        print(f"El archivo '{nombre_fichero}' no existe.")
        return None
    else:
        print("Lectura realizada correctamente.")
        return linea
    finally:
        if f is not None:
            f.close()
            # print("Archivo cerrado.")  # opcional

linea = leer_linea_segura("datos.txt")
if linea is not None:
    print("Primera línea:", linea)
```

Solución Ejercicio 19

```
try:
    n = float(input("Introduce un número: "))
    if n < 0:
        raise ValueError("El número no puede ser negativo.")
except ValueError as e:
    print("Error:", e)
else:
    print("Número válido:", n)
```

Solución Ejercicio 20

```
try:
    op = input("Operación (+, -, *, /): ")
    a = float(input("Introduce el primer número: "))
    b = float(input("Introduce el segundo número: "))

    if op == "+":
        resultado = a + b
    elif op == "-":
        resultado = a - b
    elif op == "*":
        resultado = a * b
    elif op == "/":
        if b == 0:
            raise ZeroDivisionError("No se puede dividir entre
cero.")
        resultado = a / b
    else:
        raise ValueError("Operación no reconocida.")

except ValueError as e:
    print("Error de valor:", e)
except ZeroDivisionError as e:
    print("Error de división:", e)
else:
    print("Resultado:", resultado)
```

CAPÍTULO 16. COLOR EN CONSOLA EN PYTHON

16.0 Introducción al color en consola

Cuando ejecutas programas en la **consola** (terminal), por defecto el texto suele aparecer en un color único (normalmente blanco o gris sobre fondo negro). Sin embargo, muchas consolas soportan **códigos especiales** que permiten:

- Cambiar el **color del texto**.
- Cambiar el **color de fondo**.
- Aplicar **estilos** (negrita, subrayado, parpadeo, etc.).

Estos códigos se llaman **códigos ANSI de escape**. Python no tiene color "de serie" en `print`, pero podemos:

- Enviar manualmente esos códigos ANSI como parte de la cadena.
- Usar **librerías** que lo hacen más cómodo (por ejemplo, `colorama`, `rich`, etc.).

En este capítulo veremos:

- **16.1** Códigos ANSI.
- **16.2** Colores básicos.
- **16.3** Estilos y fondos.
- **16.4** Librerías de color.
- **16.5** Errores comunes.

Y al final tendrás **20 ejercicios con solución** para practicar.

16.1 Códigos ANSI

Los **códigos ANSI** son secuencias de caracteres que la consola interpreta como órdenes de formato, no como texto normal.

La forma general de un código ANSI de color es:

`\033[<código>m`

- `\033` es el carácter **ESC** (escape) en octal.
- `[` indica el inicio de la secuencia de control.
- `<código>` es un número (o varios separados por `;`) que indica el estilo/color.
- `m` indica que es un código de **formato de texto**.

Ejemplo mínimo en Python:

```
print("\033[31mEste texto es rojo\033[0m")
```

- `\033[31m` → activa el color rojo.
- `\033[0m` → **resetea** el formato (vuelve a lo normal).

Es muy importante **resetear** al final, para no dejar la consola "enganchada" en un color.

Estructura típica en Python

```
RESET = "\033[0m"
ROJO = "\033[31m"

print(ROJO + "Texto en rojo" + RESET)
print("Texto normal")
```

16.2 Colores básicos

Los códigos ANSI más usados para **color de texto** (foreground) son:

- **30**: negro
- **31**: rojo
- **32**: verde
- **33**: amarillo
- **34**: azul
- **35**: magenta
- **36**: cian
- **37**: blanco (claro)

Ejemplo:

```
NEGRO = "\033[30m"
ROJO = "\033[31m"
VERDE = "\033[32m"
AMARILLO = "\033[33m"
AZUL = "\033[34m"
MAGENTA = "\033[35m"
CIAN = "\033[36m"
BLANCO = "\033[37m"
RESET = "\033[0m"

print(ROJO + "Texto en rojo" + RESET)
print(VERDE + "Texto en verde" + RESET)
print(AZUL + "Texto en azul" + RESET)
```

Ejemplo: mostrar todos los colores básicos

```
RESET = "\033[0m"
```

```
colores = {
    "NEGRO": "\033[30m",
    "ROJO": "\033[31m",
    "VERDE": "\033[32m",
    "AMARILLO": "\033[33m",
    "AZUL": "\033[34m",
    "MAGENTA": "\033[35m",
    "CIAN": "\033[36m",
    "BLANCO": "\033[37m",
}

for nombre, codigo in colores.items():
    print(codigo + nombre + RESET)
```

16.3 Estilos y fondos

Además del color del texto, podemos aplicar **estilos** y **colores de fondo**.

Estilos básicos

Algunos códigos de estilo (no todos funcionan igual en todas las consolas):

- **0**: reset (restablece todo).
- **1**: negrita (bold).
- **2**: tenue (dim).
- **4**: subrayado (underline).
- **5**: parpadeo (blink).
- **7**: inverso (invierte colores de texto y fondo).

Ejemplo de negrita y subrayado:

```
RESET = "\033[0m"
NEGRITA = "\033[1m"
SUBRAYADO = "\033[4m"

print(NEGRITA + "Texto en negrita" + RESET)
print(SUBRAYADO + "Texto subrayado" + RESET)
print(NEGRITA + SUBRAYADO + "Texto negrita y subrayado" + RESET)
```

Colores de fondo

Los códigos de **fondo** (background) básicos son:

- **40**: fondo negro
- **41**: fondo rojo
- **42**: fondo verde
- **43**: fondo amarillo
- **44**: fondo azul
- **45**: fondo magenta
- **46**: fondo cian
- **47**: fondo blanco

Ejemplo:

```
RESET = "\033[0m"
FONDO_ROJO = "\033[41m"
FONDO_VERDE = "\033[42m"
FONDO_AZUL = "\033[44m"

print(FONDO_ROJO + "Fondo rojo" + RESET)
print(FONDO_VERDE + "Fondo verde" + RESET)
print(FONDO_AZUL + "Fondo azul" + RESET)
```

Combinando estilos, texto y fondo

Podemos combinar varios códigos separándolos con `;`.

Ejemplo: texto amarillo, fondo azul, negrita:

```
RESET = "\033[0m"
ESTILO = "\033[1;33;44m"  # 1=negrita, 33=amarillo, 44= fondo azul

print(ESTILO + "Texto amarillo, fondo azul, negrita" + RESET)
```

Otro ejemplo:

```
RESET = "\033[0m"

print("\033[1;31mNegrita roja\033[0m")
print("\033[4;32mSubrayado verde\033[0m")
print("\033[7;34mInverso azul\033[0m")
```

16.4 Librerías de color

Escribir códigos ANSI a mano puede ser:

- Poco legible.

- Propenso a errores.
- Menos portable (en Windows clásico, por ejemplo, antes no se veían bien).

Por eso existen librerías que facilitan el uso de color.

16.4.1 Colorama

Colorama es una librería muy popular para color en consola, especialmente en Windows.

Instalación (en tu terminal):

```
pip install colorama
```

Uso básico:

```
from colorama import init, Fore, Back, Style

init(autoreset=True)

print(Fore.RED + "Texto rojo")
print(Fore.GREEN + "Texto verde")
print(Back.YELLOW + "Fondo amarillo")
print(Style.BRIGHT + "Texto brillante (negrita)")
print(Fore.BLUE + Back.WHITE + "Azul sobre blanco")
```

- `init(autoreset=True)` hace que no tengas que poner manualmente el reset cada vez.
- `Fore` → colores de texto.
- `Back` → colores de fondo.
- `Style` → estilos (BRIGHT, DIM, NORMAL, RESET_ALL).

Colores disponibles en `Fore` y `Back`:

- `BLACK`, `RED`, `GREEN`, `YELLOW`, `BLUE`, `MAGENTA`, `CYAN`, `WHITE`, `RESET`.

Ejemplo: menú coloreado con `colorama`:

```
from colorama import init, Fore, Style

init(autoreset=True)

print(Style.BRIGHT + Fore.CYAN + "=== MENÚ PRINCIPAL ===")
print(Fore.GREEN + "1. Opción uno")
print(Fore.YELLOW + "2. Opción dos")
print(Fore.RED + "3. Salir")
```

16.4.2 Otras librerías (mención rápida)

- **rich**: para salidas muy avanzadas (tablas, barras de progreso, etc.).
- **termcolor**: otra librería sencilla para colorear texto.

Ejemplo muy breve con `termcolor` (si la instalas):

```
pip install termcolor
```

```
from termcolor import colored


print(colored("Texto rojo sobre fondo blanco", "red", "on_white"))
print(colored("Texto verde subrayado", "green",
attrs=["underline"]))
```

16.5 Errores comunes con color en consola

Trabajar con color en consola es sencillo, pero hay errores típicos:

1. Olvidar el reset (`\033[0m`)

- Si no reseteas, el color/estilo se “arrastra” a las siguientes líneas.
- Solución: definir una constante `RESET` y usarla siempre al final.

2. Usar códigos ANSI en consolas que no los soportan

- Algunas consolas antiguas o entornos especiales pueden no interpretar los códigos.
- En Windows moderno suele funcionar, pero en versiones antiguas era problemático (de ahí `colorama`).

3. Mezclar demasiados códigos sin orden

- Es fácil equivocarse al escribir `\033[1;31;44m` y poner un número mal.
- Solución: definir constantes claras y reutilizarlas.

4. No pensar en la legibilidad

- Colores muy chillones o combinaciones de texto/fondo con poco contraste dificultan la lectura.
- Recuerda que el objetivo del color es **ayudar**, no molestar.

5. Depender del color para la lógica

- El color es solo presentación. No debe ser necesario para entender la salida.
- Siempre que uses color, asegúrate de que el texto sigue siendo comprensible sin él.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Escribe un programa que muestre la frase `"Hola, mundo en color"` en rojo usando códigos ANSI, y luego vuelva al color normal.

Ejercicio 2:
Crea constantes para los colores básicos (negro, rojo, verde, amarillo, azul, magenta, cian, blanco) y muestra una línea con el nombre de cada color en su color correspondiente.

Ejercicio 3:
Escribe un programa que muestre un pequeño “aviso de error” en rojo y negrita, por ejemplo: `"ERROR: archivo no encontrado"`.

Ejercicio 4:
Muestra un texto con fondo verde y texto negro que diga `"OPERACIÓN CORRECTA"`.

Ejercicio 5:
Crea un pequeño menú en consola donde el título esté en cian y negrita, las opciones en blanco normal y la opción de salir en rojo.

Ejercicio 6:
Escribe un programa que pida al usuario un número del 1 al 3 y muestre un mensaje en un color distinto según el número (por ejemplo, 1=verde, 2=amarillo, 3=rojo).

Ejercicio 7:
Crea una función `color_texto(texto, color)` que reciba una cadena y un código de color ANSI (por ejemplo, `"\033[31m"`) y devuelva el texto coloreado con reset al final. Pruébala con varios colores.

Ejercicio 8:
Crea una función `resaltar(texto)` que devuelva el texto en amarillo y negrita. Úsala para mostrar varios mensajes importantes.

Ejercicio 9:
Escribe un programa que muestre una “barra de progreso” simple usando # y que el texto de la barra esté en verde. (No hace falta que sea dinámica, solo imprimir una línea).

Ejercicio 10:
Instala `colorama` y escribe un programa que muestre tres líneas: una en rojo, otra en verde y otra en azul, usando `Fore`.

Ejercicio 11:
Con `colorama`, crea un menú donde el título esté en `Style.BRIGHT` y `Fore.CYAN`, y las opciones en `Fore.YELLOW`.

Ejercicio 12:
Escribe un programa que muestre un mensaje de “éxito” en verde y un mensaje de “error” en rojo, usando `colorama` y `autoreset=True`.

Ejercicio 13:
Crea una pequeña “tabla” de tres filas y tres columnas, donde la cabecera esté en negrita y cian, y los datos en blanco normal. Puedes usar `print` con tabulaciones o espacios.

Ejercicio 14:
Escribe un programa que muestre un texto con estilo inverso (código 7) y comprueba cómo se ve en tu consola.

Ejercicio 15:
Crea una función `imprimir_error(mensaje)` que muestre el mensaje en rojo y negrita, y otra función `imprimir_ok(mensaje)` que lo muestre en verde. Úsalas en un pequeño programa que simule validaciones.

Ejercicio 16:
Escribe un programa que muestre un texto subrayado en azul.

Ejercicio 17:
Crea un programa que imprima una lista de tareas, donde las tareas "completadas" aparezcan en verde y las "pendientes" en amarillo. Puedes usar una lista de diccionarios con `{"texto": ..., "completada": True/False}`.

Ejercicio 18:
Usando `colorama`, crea un programa que pida al usuario su nombre y luego lo salude con un mensaje en cian brillante.

Ejercicio 19:
Escribe un programa que muestre un mensaje en rojo sin resetear el color, y luego otro mensaje. Observa el efecto. Después corrige el programa añadiendo el reset para que el segundo mensaje salga en color normal.

Ejercicio 20:
Crea un pequeño "panel de estado" donde se muestren tres líneas:

- `"ESTADO: OK"` en verde.
- `"ADVERTENCIA: uso de memoria alto"` en amarillo.
- `"ERROR: conexión perdida"` en rojo.
  Hazlo primero con códigos ANSI y luego con `colorama`.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
RESET = "\033[0m"
ROJO = "\033[31m"


print(ROJO + "Hola, mundo en color" + RESET)
```

Solución Ejercicio 2

```
RESET = "\033[0m"
NEGRO = "\033[30m"
ROJO = "\033[31m"
VERDE = "\033[32m"
```

```
AMARILLO = "\033[33m"
AZUL = "\033[34m"
MAGENTA = "\033[35m"
CIAN = "\033[36m"
BLANCO = "\033[37m"

print(NEGRO + "NEGRO" + RESET)
print(ROJO + "ROJO" + RESET)
print(VERDE + "VERDE" + RESET)
print(AMARILLO + "AMARILLO" + RESET)
print(AZUL + "AZUL" + RESET)
print(MAGENTA + "MAGENTA" + RESET)
print(CIAN + "CIAN" + RESET)
print(BLANCO + "BLANCO" + RESET)
```

Solución Ejercicio 3

```
RESET = "\033[0m"
ROJO_NEGRITA = "\033[1;31m"

print(ROJO_NEGRITA + "ERROR: archivo no encontrado" + RESET)
```

Solución Ejercicio 4

```
RESET = "\033[0m"
FONDO_VERDE = "\033[42m"
NEGRO = "\033[30m"

print(FONDO_VERDE + NEGRO + "OPERACIÓN CORRECTA" + RESET)
```

Solución Ejercicio 5

```
RESET = "\033[0m"
TITULO = "\033[1;36m"   # negrita + cian
BLANCO = "\033[37m"
```

```
ROJO = "\033[31m"

print(TITULO + "=== MENÚ PRINCIPAL ===" + RESET)
print(BLANCO + "1. Opción uno" + RESET)
print(BLANCO + "2. Opción dos" + RESET)
print(ROJO + "3. Salir" + RESET)
```

Solución Ejercicio 6

```
RESET = "\033[0m"
VERDE = "\033[32m"
AMARILLO = "\033[33m"
ROJO = "\033[31m"

op = input("Introduce un número (1-3): ")

if op == "1":
    print(VERDE + "Has elegido la opción 1" + RESET)
elif op == "2":
    print(AMARILLO + "Has elegido la opción 2" + RESET)
elif op == "3":
    print(ROJO + "Has elegido la opción 3" + RESET)
else:
    print("Opción no válida")
```

Solución Ejercicio 7

```
RESET = "\033[0m"

def color_texto(texto, color):
    return color + texto + RESET

ROJO = "\033[31m"
VERDE = "\033[32m"
```

```
AZUL = "\033[34m"

print(color_texto("Texto en rojo", ROJO))
print(color_texto("Texto en verde", VERDE))
print(color_texto("Texto en azul", AZUL))
```

Solución Ejercicio 8

```
RESET = "\033[0m"
AMARILLO_NEGRITA = "\033[1;33m"

def resaltar(texto):
    return AMARILLO_NEGRITA + texto + RESET

print(resaltar("ATENCIÓN: Este es un mensaje importante"))
print(resaltar("CUIDADO: Cambios no guardados"))
```

Solución Ejercicio 9

```
RESET = "\033[0m"
VERDE = "\033[32m"

barra = "#" * 20  # barra de 20 caracteres
print(VERDE + "[" + barra + "] 100%" + RESET)
```

Solución Ejercicio 10

```
from colorama import init, Fore

init(autoreset=True)

print(Fore.RED + "Línea en rojo")
print(Fore.GREEN + "Línea en verde")
print(Fore.BLUE + "Línea en azul")
```

Solución Ejercicio 11

```
from colorama import init, Fore, Style

init(autoreset=True)

print(Style.BRIGHT + Fore.CYAN + "=== MENÚ PRINCIPAL ===")
print(Fore.YELLOW + "1. Opción uno")
print(Fore.YELLOW + "2. Opción dos")
print(Fore.YELLOW + "3. Salir")
```

Solución Ejercicio 12

```
from colorama import init, Fore

init(autoreset=True)

print(Fore.GREEN + "ÉXITO: La operación se realizó
correctamente.")
print(Fore.RED + "ERROR: No se pudo completar la operación.")
```

Solución Ejercicio 13

```
RESET = "\033[0m"
CABECERA = "\033[1;36m"  # negrita + cian
BLANCO = "\033[37m"

print(CABECERA + "Nombre\tEdad\tCiudad" + RESET)
print(BLANCO + "Ana\t30\tMadrid" + RESET)
print(BLANCO + "Luis\t25\tZaragoza" + RESET)
print(BLANCO + "Marta\t40\tValencia" + RESET)
```

Solución Ejercicio 14

```
RESET = "\033[0m"
INVERSO = "\033[7m"
```

```
print(INVERSO + "Texto con colores invertidos" + RESET)
```

Solución Ejercicio 15

```
RESET = "\033[0m"
ROJO_NEGRITA = "\033[1;31m"
VERDE = "\033[32m"

def imprimir_error(mensaje):
    print(ROJO_NEGRITA + "ERROR: " + mensaje + RESET)

def imprimir_ok(mensaje):
    print(VERDE + "OK: " + mensaje + RESET)

imprimir_ok("Datos guardados correctamente.")
imprimir_error("No se pudo conectar al servidor.")
```

Solución Ejercicio 16

```
RESET = "\033[0m"
AZUL_SUBRAYADO = "\033[4;34m"  # subrayado + azul

print(AZUL_SUBRAYADO + "Texto subrayado en azul" + RESET)
```

Solución Ejercicio 17

```
RESET = "\033[0m"
VERDE = "\033[32m"
AMARILLO = "\033[33m"

tareas = [
    {"texto": "Comprar pan", "completada": True},
    {"texto": "Estudiar Python", "completada": False},
    {"texto": "Hacer ejercicio", "completada": True},
```

```
    {"texto": "Llamar al dentista", "completada": False},
]

for t in tareas:
    if t["completada"]:
        print(VERDE + "[X] " + t["texto"] + RESET)
    else:
        print(AMARILLO + "[ ] " + t["texto"] + RESET)
```

Solución Ejercicio 18

```
from colorama import init, Fore, Style

init(autoreset=True)

nombre = input("Introduce tu nombre: ")
print(Style.BRIGHT + Fore.CYAN + f"Hola, {nombre}, ¡bienvenido a la consola en color!")
```

Solución Ejercicio 19

Versión incorrecta (sin reset):

```
ROJO = "\033[31m"

print(ROJO + "Este mensaje es rojo")
print("Este mensaje también se verá rojo (sin reset)")
```

Versión corregida:

```
ROJO = "\033[31m"
RESET = "\033[0m"

print(ROJO + "Este mensaje es rojo" + RESET)
print("Este mensaje ya es normal")
```

Solución Ejercicio 20

Versión con códigos ANSI:

```
RESET = "\033[0m"
VERDE = "\033[32m"
AMARILLO = "\033[33m"
ROJO = "\033[31m"

print(VERDE + "ESTADO: OK" + RESET)
print(AMARILLO + "ADVERTENCIA: uso de memoria alto" + RESET)
print(ROJO + "ERROR: conexión perdida" + RESET)
```

Versión con colorama:

```
from colorama import init, Fore

init(autoreset=True)

print(Fore.GREEN + "ESTADO: OK")
print(Fore.YELLOW + "ADVERTENCIA: uso de memoria alto")
print(Fore.RED + "ERROR: conexión perdida")
```

CAPÍTULO 17. FORMATO DE TEXTO Y NÚMEROS EN PYTHON

17.0 Introducción al formato

En programación, **formatear** significa presentar la información de manera clara, ordenada y legible:

- Mostrar textos con datos insertados (nombre, edad, fecha…).
- Alinear columnas en tablas.
- Controlar decimales, porcentajes, notación científica.
- Rellenar con espacios o ceros.

En Python, las dos herramientas modernas y recomendadas para formatear son:

- **`str.format()`** (método `format()`).
- **f-strings** (literal de cadena con `f"..."`).

En este capítulo veremos:

- **17.1** `format()`
- **17.2** f-strings
- **17.3** Alineación
- **17.4** Relleno
- **17.5** Formato numérico
- **17.6** Errores típicos

Con muchos ejemplos y, al final, **20 ejercicios con solución**.

17.1 El método format()

El método **`format()`** se aplica sobre una cadena y permite insertar valores en posiciones marcadas por `{}`.

Sintaxis básica

```
texto = "Hola, {}. Tienes {} años."
resultado = texto.format("Luis", 25)
print(resultado)
```

Salida:

```
Hola, Luis. Tienes 25 años.
```

- Los `{}` son **marcadores de posición**.
- `format("Luis", 25)` inserta los valores en orden.

Posiciones por índice

Podemos indicar el índice del argumento:

```
texto = "Primero: {0}, segundo: {1}, otra vez primero: {0}"
```

```
resultado = texto.format("A", "B")
print(resultado)
```

Salida:

```
Primero: A, segundo: B, otra vez primero: A
```

Nombres de parámetros

También podemos usar **nombres** en lugar de índices:

```
texto = "Nombre: {nombre}, Edad: {edad}"
resultado = texto.format(nombre="Ana", edad=30)
print(resultado)
```

Salida:

```
Nombre: Ana, Edad: 30
```

Esto es muy útil cuando hay muchos parámetros y queremos claridad.

Mezclar índices y nombres (no recomendado)

Se puede mezclar, pero es mejor evitarlo para no confundir:

```
texto = "Usuario: {0}, Nombre: {nombre}"
resultado = texto.format("user01", nombre="Ana")
print(resultado)
```

17.2 f-strings (formatted string literals)

Las **f-strings** son la forma más moderna y cómoda de formatear en Python (desde la versión 3.6).

Se escriben anteponiendo una `f` a la cadena: `f"..."`. Dentro de `{}` se pueden poner **expresiones de Python**.

Ejemplo básico

```
nombre = "Luis"
edad = 25

texto = f"Hola, {nombre}. Tienes {edad} años."
print(texto)
```

Salida:

```
Hola, Luis. Tienes 25 años.
```

Expresiones dentro de f-strings

Dentro de `{}` podemos poner operaciones:

```
a = 10
b = 3


print(f"{a} + {b} = {a + b}")
print(f"{a} / {b} = {a / b}")
```

Salida:

```
10 + 3 = 13
10 / 3 = 3.3333333333333335
```

f-strings con funciones

```
def cuadrado(x):
    return x * x


n = 5
print(f"El cuadrado de {n} es {cuadrado(n)}")
```

Comparación rápida con format()

Con `format()`:

```
nombre = "Ana"
edad = 30
texto = "Hola, {}. Tienes {} años.".format(nombre, edad)
```

Con f-string:

```
texto = f"Hola, {nombre}. Tienes {edad} años."
```

Las f-strings suelen ser más **legibles** y **cortas**.

17.3 Alineación

Tanto `format()` como f-strings permiten especificar **alineación** dentro de un campo de ancho fijo.

La sintaxis general dentro de `{}` es:

```
{valor:formato}
```

Donde `formato` puede incluir:

- **ancho** (número de caracteres).
- **símbolo de alineación** (<, >, ^).

Alineación a la izquierda <

```
texto = "{:<10}".format("Ana")
print(texto)
print(len(texto))
```

Salida (visual):

```
Ana
10
```

- `:<10` → ancho 10, alineado a la izquierda.

Con f-string:

```
nombre = "Ana"
print(f"{nombre:<10}")
```

Alineación a la derecha >

```
texto = "{:>10}".format("Ana")
print(texto)
print(len(texto))
```

Salida:

```
       Ana
10
```

Con f-string:

```
nombre = "Ana"
print(f"{nombre:>10}")
```

Alineación centrada ^

```
texto = "{:^10}".format("Ana")
print(texto)
print(len(texto))
```

Salida (aprox.):

```
   Ana
10
```

Con f-string:

```
nombre = "Ana"
print(f"{nombre:^10}")
```

Ejemplo: tabla alineada

```
cabecera = f"{'Nombre':<10}{'Edad':>5}"
fila1 = f"{'Ana':<10}{30:>5}"
fila2 = f"{'Luis':<10}{25:>5}"

print(cabecera)
print(fila1)
print(fila2)
```

Salida:

```
Nombre       Edad
Ana           30
Luis          25
```

17.4 Relleno

Además de alinear, podemos **rellenar** con un carácter distinto del espacio.

La sintaxis es:

```
{valor:caracteralineaciónancho}
```

Donde `caracter` es el carácter de relleno.

Relleno con ceros

Muy típico en números:

```
numero = 42
print("{:05}".format(numero))   # ancho 5, relleno con ceros
```

Salida:

```
00042
```

Con f-string:

```
numero = 42
print(f"{numero:05}")
```

Relleno con otros caracteres

```
texto = "{:*^10}".format("Ana")
print(texto)
```

- `*` → carácter de relleno.
- `^` → centrado.

- `10` → ancho.

Salida:

```
***Ana****
```

Con f-string:

```
nombre = "Ana"
print(f"{nombre:*^10}")
```

Otro ejemplo:

```
print(f"{'ID':-<10}")   # relleno con '-'
print(f"{'FIN':.=10}")  # relleno con '.'
```

17.5 Formato numérico

El formato numérico es muy importante para:

- Controlar decimales.
- Mostrar porcentajes.
- Usar notación científica.
- Separadores de miles.

Decimales y precisión

Sintaxis:

```
{numero:.nf}
```

Donde `n` es el número de decimales.

Ejemplo:

```
pi = 3.1415926535


print("{:.2f}".format(pi))   # 2 decimales
print("{:.4f}".format(pi))   # 4 decimales
```

Salida:

```
3.14
3.1416
```

Con f-string:

```
print(f"{pi:.2f}")
print(f"{pi:.4f}")
```

Ancho + decimales

```
valor = 123.456
```

```
print("{:10.2f}".format(valor))   # ancho 10, 2 decimales
print(f"{valor:10.2f}")
```

Salida (aprox.):

```
    123.46
    123.46
```

Porcentajes

Podemos multiplicar por 100 y añadir el símbolo %, o usar el tipo `"%"`:

```
ratio = 0.2567

print("{:.2%}".format(ratio))   # formato porcentaje
print(f"{ratio:.2%}")
```

Salida:

```
25.67%
25.67%
```

Notación científica

Tipo `e` o `E`:

```
numero = 1234567.89

print("{:.3e}".format(numero))
print(f"{numero:.3e}")
```

Salida:

```
1.235e+06
1.235e+06
```

Separador de miles

Podemos usar `,` como separador de miles (en notación anglosajona):

```
numero = 1234567.89

print("{:,.2f}".format(numero))
print(f"{numero:,.2f}")
```

Salida:

```
1,234,567.89
```

```
1,234,567.89
```

En algunos entornos, si quieres formato europeo (punto de miles, coma decimal), tendrás que hacer transformaciones adicionales (reemplazar caracteres, etc.).

Ejemplo completo de formato numérico

```
importe = 1234.56789


print("Importe bruto:", f"{importe:10.2f}")
print("Importe en notación científica:", f"{importe:.3e}")
print("Importe con miles:", f"{importe:,.2f}")
print("Importe como porcentaje:", f"{importe/100:.1%}")
```

17.6 Errores típicos y buenas prácticas

Error 1: usar % antiguo en lugar de format/f-strings

Python permite el estilo antiguo:

```
print("Hola, %s. Tienes %d años." % ("Luis", 25))
```

Funciona, pero es menos flexible y más propenso a errores. Se recomienda usar **`format()`** o **f-strings**.

Error 2: olvidar el formato dentro de `{}`

A veces se escribe:

```
print(f"{pi:2f}")  # incorrecto
```

Debe ser:

```
print(f"{pi:.2f}")  # correcto
```

El punto `.` indica decimales.

Error 3: mezclar tipos incompatibles

Ejemplo:

```
print("{:d}".format(3.14))  # d es para enteros
```

Esto da error (`ValueError`). Para decimales, usa `f` o no pongas tipo.

Error 4: ancho insuficiente

Si el ancho es menor que el número de caracteres, Python no recorta, simplemente usa el ancho necesario.

```
print(f"{123456:3d}")  # ancho 3, pero el número tiene 6 dígitos
```

Salida:

```
123456
```

No es un error, pero puede romper la alineación esperada.

Error 5: olvidar que f-strings evalúan expresiones

Dentro de `{}` se evalúa código. Si pones algo incorrecto, tendrás error de sintaxis o de ejecución.

```
print(f"{x+}")  # error de sintaxis
```

Buenas prácticas

- **Usar f-strings** para la mayoría de casos: son claras y potentes.
- **Usar nombres descriptivos** en `format()` cuando haya muchos parámetros.
- **Definir plantillas** para tablas y reutilizarlas.
- **Probar el formato** con ejemplos pequeños antes de usarlo en grandes informes.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Escribe una cadena con `format()` que diga: `"Hola, NOMBRE. Tienes EDAD años."` usando dos parámetros posicionales.

Ejercicio 2:
Repite el ejercicio anterior usando parámetros con nombre (`nombre`, `edad`).

Ejercicio 3:
Usa una f-string para mostrar: `"El resultado de A + B es R"`, donde `A`, `B` y `R` son variables.

Ejercicio 4:
Crea una pequeña tabla con tres personas (nombre, edad) usando f-strings y alineando el nombre a la izquierda en 10 caracteres y la edad a la derecha en 3 caracteres.

Ejercicio 5:
Muestra el número `3.14159265` con 2, 4 y 6 decimales usando f-strings.

Ejercicio 6:
Muestra el número `1234.567` con ancho 10 y 2 decimales, alineado a la derecha.

Ejercicio 7:
Muestra el número `0.2567` como porcentaje con 1 decimal.

Ejercicio 8:
Muestra el número `1234567.89` con separador de miles y 2 decimales.

Ejercicio 9:
Crea una f-string que muestre `"ID: 00042"` usando relleno con ceros hasta 5 dígitos.

Ejercicio 10:
Crea una f-string que muestre el texto `"TÍTULO"` centrado en un campo de 20 caracteres, rellenando con `*`.

Ejercicio 11:
Usa `format()` para mostrar `"Producto: X, Precio: Y"` donde `X` es una cadena y `Y` un número con 2 decimales.

Ejercicio 12:
Crea una f-string que muestre una operación de división con 3 decimales en el resultado, por ejemplo: `"10 / 3 = 3.333"`.

Ejercicio 13:
Crea una tabla de tres productos (nombre, precio) donde el nombre esté alineado a la izquierda en 15 caracteres y el precio a la derecha en 8 caracteres con 2 decimales.

Ejercicio 14:
Muestra el número `123456789` en notación científica con 4 decimales.

Ejercicio 15:
Crea una f-string que muestre `"Temperatura: X °C"` donde `X` tenga 1 decimal.

Ejercicio 16:
Usa `format()` para mostrar `"Hora: HH:MM"` donde `HH` y `MM` son enteros rellenados con ceros hasta 2 dígitos.

Ejercicio 17:
Crea una f-string que muestre `"Saldo: 1.234,57 €"` partiendo del número `1234.567`, usando formato con miles y 2 decimales, y luego reemplazando `.` y `,` para formato europeo.

Ejercicio 18:
Escribe una f-string que muestre una lista de números `[1, 2, 3]` y su suma, todo en una misma frase.

Ejercicio 19:
Crea una plantilla con `format()` para una línea de log: `"[NIVEL] Mensaje: TEXTO"`, y úsala con distintos niveles (`INFO`, `ERROR`, `DEBUG`).

Ejercicio 20:
Crea una pequeña “tabla de notas” con tres alumnos (nombre, nota media) donde la nota se muestre con 1 decimal y todo esté alineado en columnas.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
texto = "Hola, {}. Tienes {} años."
resultado = texto.format("Luis", 25)
print(resultado)
```

Solución Ejercicio 2

```
texto = "Hola, {nombre}. Tienes {edad} años."
```

```
resultado = texto.format(nombre="Ana", edad=30)
print(resultado)
```

Solución Ejercicio 3

```
A = 10
B = 3
R = A + B

print(f"El resultado de {A} + {B} es {R}")
```

Solución Ejercicio 4

```
personas = [
    ("Ana", 30),
    ("Luis", 25),
    ("Marta", 40),
]

cabecera = f"{'Nombre':<10}{'Edad':>3}"
print(cabecera)
for nombre, edad in personas:
    print(f"{nombre:<10}{edad:>3}")
```

Solución Ejercicio 5

```
pi = 3.14159265

print(f"{pi:.2f}")
print(f"{pi:.4f}")
print(f"{pi:.6f}")
```

Solución Ejercicio 6

```
numero = 1234.567
print(f"{numero:10.2f}")
```

Solución Ejercicio 7

```
ratio = 0.2567
print(f"{ratio:.1%}")
```

Solución Ejercicio 8

```
numero = 1234567.89
print(f"{numero:,.2f}")
```

Solución Ejercicio 9

```
id_num = 42
print(f"ID: {id_num:05}")
```

Salida:

```
ID: 00042
```

Solución Ejercicio 10

```
titulo = "TÍTULO"
print(f"{titulo:*^20}")
```

Ejemplo de salida:

```
*******TÍTULO********
```

Solución Ejercicio 11

```
producto = "Teclado"
precio = 25.5

texto = "Producto: {0}, Precio: {1:.2f}"
resultado = texto.format(producto, precio)
print(resultado)
```

Solución Ejercicio 12

```
a = 10
b = 3
```

```
resultado = a / b

print(f"{a} / {b} = {resultado:.3f}")
```

Solución Ejercicio 13

```
productos = [
    ("Teclado", 25.5),
    ("Ratón", 15.99),
    ("Monitor", 199.9),
]

cabecera = f"{'Producto':<15}{'Precio':>8}"
print(cabecera)
for nombre, precio in productos:
    print(f"{nombre:<15}{precio:>8.2f}")
```

Solución Ejercicio 14

```
numero = 123456789
print(f"{numero:.4e}")
```

Solución Ejercicio 15

```
temp = 23.456
print(f"Temperatura: {temp:.1f} °C")
```

Solución Ejercicio 16

```
hora = 9
minuto = 5

texto = "Hora: {:02d}:{:02d}"
resultado = texto.format(hora, minuto)
print(resultado)
```

Salida:

```
Hora: 09:05
```

Solución Ejercicio 17

```
saldo = 1234.567


# Formato anglosajón con miles y decimales
formato = f"{saldo:,.2f}"  # '1,234.57'


# Convertir a formato europeo: punto de miles, coma decimal
formato_europeo = formato.replace(",", "X").replace(".", ",").replace("X", ".")
print(f"Saldo: {formato_europeo} €")
```

Solución Ejercicio 18

```
numeros = [1, 2, 3]
suma = sum(numeros)


print(f"La lista es {numeros} y su suma es {suma}.")
```

Solución Ejercicio 19

```
plantilla = "[{nivel}] Mensaje: {texto}"


print(plantilla.format(nivel="INFO", texto="Inicio del programa"))
print(plantilla.format(nivel="ERROR", texto="Archivo no encontrado"))
print(plantilla.format(nivel="DEBUG", texto="Valor de x=42"))
```

Solución Ejercicio 20

```
alumnos = [
    ("Ana", 8.75),
    ("Luis", 6.5),
    ("Marta", 9.2),
]
```

```
cabecera = f"{'Alumno':<10}{'Nota':>6}"
print(cabecera)
for nombre, nota in alumnos:
    print(f"{nombre:<10}{nota:>6.1f}")
```

CAPÍTULO 18. MÓDULOS EN PYTHON

18.0 Introducción a los módulos

En Python, un **módulo** es simplemente un **fichero `.py`** que contiene código: funciones, clases, variables, constantes, etc. Los módulos permiten:

- **Organizar** el código en partes más pequeñas y manejables.
- **Reutilizar** funciones y clases en distintos programas.
- **Compartir** código entre proyectos.
- Usar la enorme **librería estándar** de Python (módulos ya hechos).

Trabajar con módulos es esencial para pasar de programas pequeños a proyectos más grandes y profesionales.

En este capítulo veremos:

- **18.1** `import`
- **18.2** `from … import`
- **18.3** Creación de módulos propios
- **18.4** Librerías estándar
- **18.5** Errores comunes

Con muchos ejemplos y, al final, **20 ejercicios con solución**.

18.1 La instrucción import

La forma más básica de usar un módulo es con la instrucción **`import`**.

Importar un módulo completo

Ejemplo clásico: el módulo **`math`** (matemáticas).

```
import math


print(math.pi)
print(math.sqrt(16))
print(math.sin(math.pi / 2))
```

- `import math` carga el módulo.
- Para usar algo de `math`, escribimos `math.nombre_función` o `math.nombre_constante`.

Importar con alias (as)

Podemos dar un **nombre corto** al módulo:

```
import math as m
```

```
print(m.pi)
print(m.sqrt(25))
```

Esto es útil cuando el nombre del módulo es largo o se usa mucho.

Importar varios módulos

```
import math
import random


print(math.sqrt(9))
print(random.randint(1, 10))
```

O en una sola línea:

```
import math, random
```

Aunque por claridad, muchas veces se prefiere una línea por módulo.

¿Qué hace realmente import?

Cuando escribes `import nombre_modulo`:

1. Python busca el módulo en:
    - El directorio actual.
    - La ruta de instalación de Python.
    - Otros directorios configurados en `sys.path`.
2. Si lo encuentra, **ejecuta** el código del módulo (una sola vez).
3. Crea un objeto módulo y lo asigna al nombre `nombre_modulo`.

Por eso, si un módulo tiene código en la parte superior (no dentro de funciones), ese código se ejecuta al hacer `import`.

18.2 from … import

Otra forma de importar es usar **`from módulo import nombre`**, que permite traer directamente funciones, clases o variables concretas.

Importar nombres específicos

```
from math import sqrt, pi


print(pi)
print(sqrt(16))
```

Aquí no usamos `math.pi` ni `math.sqrt`, sino directamente `pi` y `sqrt`.

Importar con alias

```
from math import sqrt as raiz
```

```
print(raiz(25))
```

Importar todo (no recomendado)

```
from math import *
```

Esto importa **todos** los nombres públicos del módulo `math` en el espacio de nombres actual.

Problemas:

- Puede **sobrescribir** nombres que ya tengas.
- Hace más difícil saber de dónde viene cada función.

Por eso, se recomienda **evitar** `from módulo import *` en código serio.

Comparación `import` vs `from … import`

Con `import math`:

```
import math


print(math.sqrt(16))
print(math.pi)
```

Con `from math import sqrt, pi`:

```
from math import sqrt, pi


print(sqrt(16))
print(pi)
```

Ambas formas son válidas. En proyectos grandes, se suele preferir:

- `import módulo` para dejar claro de dónde viene cada cosa.
- `from módulo import nombre` cuando se usan pocas funciones y se quiere código más limpio.

18.3 Creación de módulos propios

La parte más interesante: **tú puedes crear tus propios módulos**.

Crear un módulo simple

Supongamos que quieres agrupar funciones matemáticas propias. Creas un fichero llamado `mis_matematicas.py` con este contenido:

```
# mis_matematicas.py


def cuadrado(x):
```

```
    return x * x

def cubo(x):
    return x * x * x

PI_PERSONAL = 3.14159
```

Ahora, en otro fichero (por ejemplo, `programa.py`) puedes usar ese módulo:

```
# programa.py

import mis_matematicas

print(mis_matematicas.cuadrado(5))
print(mis_matematicas.cubo(2))
print(mis_matematicas.PI_PERSONAL)
```

Importante: ambos ficheros (`mis_matematicas.py` y `programa.py`) deben estar en el **mismo directorio** (o en un directorio que Python conozca).

Usar from … import con módulos propios

```
from mis_matematicas import cuadrado, PI_PERSONAL

print(cuadrado(7))
print(PI_PERSONAL)
```

Variables especiales: name

Dentro de un módulo, existe la variable especial **`__name__`**.

- Si el módulo se ejecuta directamente (por ejemplo, `python mis_matematicas.py`), entonces `__name__ == "__main__"`.
- Si el módulo se importa desde otro, `__name__` será el nombre del módulo (`"mis_matematicas"`).

Esto permite escribir código que solo se ejecute cuando el módulo se ejecuta directamente, no cuando se importa:

```
# mis_matematicas.py

def cuadrado(x):
    return x * x
```

```
def cubo(x):
    return x * x * x


if __name__ == "__main__":
    # Código de prueba
    print("Pruebas de mis_matematicas:")
    print("cuadrado(3) =", cuadrado(3))
    print("cubo(2) =", cubo(2))
```

Si ejecutas `python mis_matematicas.py`, verás las pruebas.
Si importas `mis_matematicas` desde otro programa, las pruebas **no** se ejecutan.

Organización en varios módulos

En proyectos más grandes, puedes tener:

- `utilidades_texto.py`
- `utilidades_numeros.py`
- `gestor_ficheros.py`
- etc.

Cada uno con funciones relacionadas. Luego, desde tu programa principal, importas lo que necesites.

18.4 Librerías estándar

Python viene con una **librería estándar** enorme: decenas de módulos ya hechos para tareas comunes. Algunos muy usados:

- **`math`**: funciones matemáticas.
- **`random`**: generación de números aleatorios.
- **`os`**: operaciones del sistema operativo (rutas, directorios…).
- **`sys`**: acceso a información del intérprete, argumentos de línea de comandos.
- **`datetime`**: fechas y horas.
- **`json`**: trabajar con JSON.
- **`csv`**: trabajar con ficheros CSV.
- **`pathlib`**: manejo de rutas de forma orientada a objetos.

Ejemplos rápidos

math:

```
import math
```

```
print("pi =", math.pi)
print("sqrt(16) =", math.sqrt(16))
print("cos(0) =", math.cos(0))
```

random:

```
import random

print("Número aleatorio entre 1 y 10:", random.randint(1, 10))
print("Número aleatorio entre 0 y 1:", random.random())
```

os:

```
import os

print("Directorio actual:", os.getcwd())
print("Contenido del directorio actual:", os.listdir("."))
```

datetime:

```
from datetime import datetime

ahora = datetime.now()
print("Fecha y hora actuales:", ahora)
print("Año:", ahora.year)
print("Mes:", ahora.month)
print("Día:", ahora.day)
```

json:

```
import json

datos = {"nombre": "Luis", "edad": 25}
cadena = json.dumps(datos)
print("JSON:", cadena)
```

La librería estándar es uno de los grandes puntos fuertes de Python: muchas tareas ya están resueltas.

18.5 Errores comunes con módulos

Trabajar con módulos es sencillo, pero hay errores típicos:

1. Nombre de fichero igual que un módulo estándar

Si llamas a tu fichero `math.py` y luego haces `import math`, Python puede intentar importar **tu fichero** en lugar del módulo estándar.

Esto puede causar errores extraños.

Buena práctica: no uses nombres de módulos estándar (`math.py`, `random.py`, `json.py`, etc.) para tus propios ficheros.

2. Módulo no encontrado (ModuleNotFoundError)

Si haces:

```
import mis_matematicas
```

y Python no encuentra `mis_matematicas.py` en el directorio actual ni en la ruta de búsqueda, obtendrás:

```
ModuleNotFoundError: No module named 'mis_matematicas'
```

Soluciones:

- Asegúrate de que el fichero está en el mismo directorio.
- Comprueba que el nombre es correcto (mayúsculas/minúsculas).

3. Importar antes de crear el módulo

Si intentas importar un módulo que aún no has creado o guardado, fallará.

4. Uso incorrecto de from … import *

`from módulo import *` puede:

- Sobrescribir nombres.
- Hacer el código menos claro.

Mejor usar `import módulo` o `from módulo import nombre`.

5. Código ejecutado al importar

Si en tu módulo tienes código que se ejecuta directamente (por ejemplo, `print()` fuera de funciones), ese código se ejecutará **cada vez que importes** el módulo.

Esto puede ser molesto si solo quieres las funciones.

Solución: usar el patrón `if __name__ == "__main__":` para código de prueba.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Escribe un programa que importe el módulo `math` y muestre el valor de `pi` y la raíz cuadrada de 81.

Ejercicio 2:
Usa `import math as m` y muestra `m.pi` y `m.sqrt(25)`.

Ejercicio 3:
Usa `from math import sqrt, pi` y muestra `pi` y `sqrt(49)`.

Ejercicio 4:
Crea un fichero `mis_funciones.py` con una función `saludar(nombre)` que imprima `"Hola, NOMBRE"`. En otro fichero, importa `mis_funciones` y llama a `saludar`.

Ejercicio 5:
Modifica `mis_funciones.py` para añadir una función `sumar(a, b)` que devuelva la suma. En el programa principal, importa solo `sumar` con `from mis_funciones import sumar` y úsala.

Ejercicio 6:
Crea un módulo `utilidades.py` con una función `es_par(n)` que devuelva `True` si `n` es par y `False` si no. Úsalo desde otro programa.

Ejercicio 7:
En `utilidades.py`, añade una función `es_primo(n)` que compruebe si un número es primo. Importa el módulo y prueba la función con varios valores.

Ejercicio 8:
Crea un módulo `mi_math.py` con funciones `cuadrado(n)` y `cubo(n)`. En el programa principal, importa el módulo y muestra el cuadrado y el cubo de 2, 3 y 4.

Ejercicio 9:
Añade al módulo `mi_math.py` un bloque `if __name__ == "__main__":` que haga pruebas de las funciones. Ejecuta `mi_math.py` directamente y observa la salida.

Ejercicio 10:
Escribe un programa que use `random.randint(1, 100)` para generar un número aleatorio y lo muestre.

Ejercicio 11:
Usa `from random import randint` para generar tres números aleatorios entre 1 y 10 y mostrarlos.

Ejercicio 12:
Escribe un programa que use `os.getcwd()` para mostrar el directorio actual y `os.listdir(".")` para listar los ficheros.

Ejercicio 13:
Usa `from datetime import datetime` para mostrar la fecha y hora actuales en el formato `"AAAA-MM-DD HH:MM:SS"`.

Ejercicio 14:
Crea un módulo `mensajes.py` con funciones `mensaje_error(texto)` y `mensaje_ok(texto)` que impriman el texto con prefijos `"ERROR:"` y `"OK:"`. Úsalo desde otro programa.

Ejercicio 15:
Intenta crear un fichero llamado `random.py` y luego haz `import random` desde otro programa. Observa el error o comportamiento extraño. Corrige el problema renombrando el fichero.

Ejercicio 16:
Crea un módulo `operaciones.py` con funciones `sumar`, `restar`, `multiplicar`, `dividir`. Importa el módulo y crea una pequeña calculadora en otro fichero.

Ejercicio 17:
Usa `from math import *` en un programa y llama a `sqrt`, `sin`, `cos` sin prefijo. Luego cambia el código para usar `import math` y `math.sqrt`, `math.sin`, `math.cos`. Compara la claridad.

Ejercicio 18:
Crea un módulo `config.py` con variables `NOMBRE_APP`, `VERSION`, `AUTOR`. En el programa principal, importa `config` y muestra un mensaje con esos datos.

Ejercicio 19:
Escribe un programa que intente importar un módulo inexistente (por ejemplo, `import modulo_que_no_existe`) y observa el error `ModuleNotFoundError`.

Ejercicio 20:
Crea un módulo `estadisticas.py` con una función `media(lista)` que calcule la media de una lista de números. Úsalo desde otro programa para calcular la media de `[1, 2, 3, 4, 5]`.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
import math


print("Valor de pi:", math.pi)
print("Raíz cuadrada de 81:", math.sqrt(81))
```

Solución Ejercicio 2

```
import math as m


print("pi =", m.pi)
print("sqrt(25) =", m.sqrt(25))
```

Solución Ejercicio 3

```
from math import sqrt, pi
```

```
print("pi =", pi)
print("sqrt(49) =", sqrt(49))
```

Solución Ejercicio 4

mis_funciones.py

```
# mis_funciones.py

def saludar(nombre):
    print(f"Hola, {nombre}")
```

programa.py

```
import mis_funciones

mis_funciones.saludar("Luis")
mis_funciones.saludar("Ana")
```

Solución Ejercicio 5

mis_funciones.py

```
# mis_funciones.py

def saludar(nombre):
    print(f"Hola, {nombre}")

def sumar(a, b):
    return a + b
```

programa.py

```
from mis_funciones import sumar

resultado = sumar(10, 5)
print("La suma es:", resultado)
```

Solución Ejercicio 6

utilidades.py

```
# utilidades.py

def es_par(n):
    return n % 2 == 0
```

programa.py

```
import utilidades

for i in range(1, 6):
    if utilidades.es_par(i):
        print(i, "es par")
    else:
        print(i, "es impar")
```

Solución Ejercicio 7

utilidades.py

```
# utilidades.py

def es_par(n):
    return n % 2 == 0

def es_primo(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

programa.py

```
import utilidades
```

```
numeros = [1, 2, 3, 4, 5, 17, 18]


for n in numeros:
    print(n, "par?" , utilidades.es_par(n), "primo?",
utilidades.es_primo(n))
```

Solución Ejercicio 8

mi_math.py

```
# mi_math.py


def cuadrado(n):
    return n * n


def cubo(n):
    return n * n * n
```

programa.py

```
import mi_math


for n in [2, 3, 4]:
    print(f"{n}^2 =", mi_math.cuadrado(n))
    print(f"{n}^3 =", mi_math.cubo(n))
```

Solución Ejercicio 9

mi_math.py

```
# mi_math.py


def cuadrado(n):
    return n * n


def cubo(n):
    return n * n * n
```

```
if __name__ == "__main__":
    print("Pruebas de mi_math:")
    print("cuadrado(3) =", cuadrado(3))
    print("cubo(2) =", cubo(2))
```

Si ejecutas `python mi_math.py`, verás las pruebas.
Si importas `mi_math` desde otro programa, las pruebas no se ejecutan.

Solución Ejercicio 10

```
import random

numero = random.randint(1, 100)
print("Número aleatorio entre 1 y 100:", numero)
```

Solución Ejercicio 11

```
from random import randint

print("Tres números aleatorios entre 1 y 10:")
print(randint(1, 10))
print(randint(1, 10))
print(randint(1, 10))
```

Solución Ejercicio 12

```
import os

print("Directorio actual:", os.getcwd())
print("Contenido del directorio actual:")
for nombre in os.listdir("."):
    print(" -", nombre)
```

Solución Ejercicio 13

```
from datetime import datetime
```

```
ahora = datetime.now()
texto = ahora.strftime("%Y-%m-%d %H:%M:%S")
print("Fecha y hora actuales:", texto)
```

Solución Ejercicio 14

mensajes.py

```
# mensajes.py

def mensaje_error(texto):
    print("ERROR:", texto)

def mensaje_ok(texto):
    print("OK:", texto)
```

programa.py

```
import mensajes

mensajes.mensaje_ok("Operación completada correctamente.")
mensajes.mensaje_error("No se pudo abrir el archivo.")
```

Solución Ejercicio 15

random.py (versión problemática):

```
# random.py

print("Este es mi módulo random (no debería llamarse así).")
```

programa.py

```
import random

print(random.randint(1, 10))  # Probablemente dará error, porque tu random.py no tiene randint
```

Solución: renombrar tu fichero a otro nombre, por ejemplo `mi_random.py`, y usar:

```
import random  # ahora importará el módulo estándar
```

Solución Ejercicio 16

operaciones.py

```
# operaciones.py

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir entre cero.")
    return a / b
```

calculadora.py

```
import operaciones

a = float(input("Introduce el primer número: "))
b = float(input("Introduce el segundo número: "))
op = input("Operación (+, -, *, /): ")

if op == "+":
    print("Resultado:", operaciones.sumar(a, b))
elif op == "-":
    print("Resultado:", operaciones.restar(a, b))
elif op == "*":
    print("Resultado:", operaciones.multiplicar(a, b))
elif op == "/":
    try:
```

```
        print("Resultado:", operaciones.dividir(a, b))
    except ValueError as e:
        print("Error:", e)
else:
    print("Operación no reconocida.")
```

Solución Ejercicio 17

Versión con `from math import *`:

```
from math import *

print("sqrt(16) =", sqrt(16))
print("sin(0) =", sin(0))
print("cos(0) =", cos(0))
```

Versión con `import math`:

```
import math

print("math.sqrt(16) =", math.sqrt(16))
print("math.sin(0) =", math.sin(0))
print("math.cos(0) =", math.cos(0))
```

La segunda versión deja más claro de dónde vienen las funciones.

Solución Ejercicio 18

config.py

```
# config.py

NOMBRE_APP = "MiAplicacion"
VERSION = "1.0"
AUTOR = "Luis"
```

programa.py

```
import config

print(f"Aplicación: {config.NOMBRE_APP}")
```

```
print(f"Versión: {config.VERSION}")
print(f"Autor: {config.AUTOR}")
```

Solución Ejercicio 19

```
import modulo_que_no_existe
```

Al ejecutar, obtendrás algo como:

```
ModuleNotFoundError: No module named 'modulo_que_no_existe'
```

Esto indica que Python no encuentra ese módulo.

Solución Ejercicio 20

estadisticas.py

```
# estadisticas.py

def media(lista):
    if len(lista) == 0:
        return 0
    return sum(lista) / len(lista)
```

programa.py

```
import estadisticas

datos = [1, 2, 3, 4, 5]
m = estadisticas.media(datos)
print("La media es:", m)
```

CAPÍTULO 19. FUNCIONES AVANZADAS EN PYTHON (CAPÍTULO AMPLIADO)

19.0 Introducción: más allá de “def”

Hasta ahora has visto funciones “normales”: las defines con `def`, les pasas parámetros, devuelven valores y listo. Pero en Python las funciones son **mucho más poderosas**:

- Son **objetos**: se pueden guardar en variables, pasar como parámetros, devolver desde otras funciones.
- Se pueden **envolver** con otras funciones para añadir comportamiento: eso son los **decoradores**.
- Pueden **recordar** valores de su entorno: eso son los **closures**.
- Se usan como base de las **funciones de orden superior** (funciones que reciben o devuelven funciones).
- Existen **buenas prácticas avanzadas** para escribir funciones limpias, reutilizables y fáciles de probar.

En este capítulo veremos:

- **19.1 Funciones como objetos**
- **19.2 Decoradores**
- **19.3 Closures**
- **19.4 Funciones de orden superior**
- **19.5 Buenas prácticas avanzadas**

Con muchos ejemplos y, al final, **20 ejercicios con solución**.

19.1 Funciones como objetos

En Python, **todo es un objeto**, y las funciones no son una excepción. Esto significa que una función:

- Se puede **asignar** a una variable.
- Se puede **pasar** como argumento a otra función.
- Se puede **devolver** desde una función.
- Se puede **guardar** en estructuras de datos (listas, diccionarios, etc.).

19.1.1 Asignar funciones a variables

```
def saludar(nombre):
    print(f"Hola, {nombre}")

# Asignamos la función a otra variable
f = saludar

# Llamamos a través de la nueva variable
```

```
f("Luis")
```

Salida:

```
Hola, Luis
```

Aquí `f` es otra referencia al mismo objeto función que `saludar`.

19.1.2 Guardar funciones en listas

```
def cuadrado(x):
    return x * x

def cubo(x):
    return x * x * x

funciones = [cuadrado, cubo]

for f in funciones:
    print(f(3))
```

Salida:

```
9
27
```

19.1.3 Funciones en diccionarios (como “switch”)

```
def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

operaciones = {
    "+": sumar,
    "-": restar,
    "*": multiplicar
```

```
}

op = input("Operación (+, -, *): ")
a = float(input("a: "))
b = float(input("b: "))

if op in operaciones:
    resultado = operaciones[op](a, b)
    print("Resultado:", resultado)
else:
    print("Operación no válida")
```

Aquí el diccionario `operaciones` mapea símbolos a funciones.

19.1.4 Pasar funciones como parámetros

```
def aplicar_operacion(a, b, operacion):
    return operacion(a, b)

def sumar(a, b):
    return a + b

def multiplicar(a, b):
    return a * b

print(aplicar_operacion(3, 4, sumar))
print(aplicar_operacion(3, 4, multiplicar))
```

Salida:

```
7
12
```

19.1.5 Devolver funciones desde funciones

```
def crear_multiplicador(factor):
    def multiplicar(x):
        return x * factor
    return multiplicar
```

```
por2 = crear_multiplicador(2)
por3 = crear_multiplicador(3)

print(por2(10))  # 20
print(por3(10))  # 30
```

Esto ya se acerca a los **closures**, que veremos en detalle en 19.3.

19.2 Decoradores

Un **decorador** es una función que **recibe una función y devuelve otra función**, normalmente añadiendo comportamiento extra sin modificar el código original.

Se usan mucho para:

- Medir tiempos de ejecución.
- Registrar (log) llamadas.
- Comprobar permisos.
- Validar parámetros.
- Cachear resultados.

19.2.1 Decorador básico “a mano”

```
def decorador(funcion):
    def nueva_funcion():
        print("Antes de llamar a la función original")
        funcion()
        print("Después de llamar a la función original")
    return nueva_funcion

def saludar():
    print("Hola")

saludar_decorada = decorador(saludar)
saludar_decorada()
```

Salida:

```
Antes de llamar a la función original
Hola
Después de llamar a la función original
```

19.2.2 Sintaxis con @decorator

Python tiene una sintaxis especial para aplicar decoradores:

```
def decorador(funcion):
    def nueva_funcion():
        print("Antes")
        funcion()
        print("Después")
    return nueva_funcion

@decorador
def saludar():
    print("Hola")

saludar()
```

Es equivalente a:

```
saludar = decorador(saludar)
```

19.2.3 Decoradores con argumentos en la función decorada

Si la función decorada recibe parámetros, el decorador debe aceptarlos y pasarlos:

```
def decorador(funcion):
    def nueva_funcion(*args, **kwargs):
        print("Llamando a la función con argumentos:", args,
kwargs)
        resultado = funcion(*args, **kwargs)
        print("La función devolvió:", resultado)
        return resultado
    return nueva_funcion

@decorador
def sumar(a, b):
    return a + b

s = sumar(3, 4)
```

```
print("Resultado final:", s)
```

19.2.4 Decoradores con parámetros (decorador de decoradores)

A veces queremos que el decorador reciba sus propios parámetros:

```
def repetir(n):
    def decorador(funcion):
        def nueva_funcion(*args, **kwargs):
            for i in range(n):
                print(f"Ejecutando vez {i+1}/{n}")
                funcion(*args, **kwargs)
        return nueva_funcion
    return decorador

@repetir(3)
def saludar():
    print("Hola")

saludar()
```

Salida:

```
Ejecutando vez 1/3
Hola
Ejecutando vez 2/3
Hola
Ejecutando vez 3/3
Hola
```

19.2.5 Decoradores y metadatos (functools.wraps)

Un problema de los decoradores es que la función resultante pierde el nombre y el docstring de la original. Para evitarlo, se usa `functools.wraps`:

```
from functools import wraps

def decorador(funcion):
    @wraps(funcion)
    def nueva_funcion(*args, **kwargs):
```

```
        print("Antes")
        resultado = funcion(*args, **kwargs)
        print("Después")
        return resultado
    return nueva_funcion

@decorador
def saludar():
    """Función que saluda."""
    print("Hola")

print(saludar.__name__)      # saludar
print(saludar.__doc__)       # Función que saluda.
```

19.3 Closures

Un **closure** (clausura) es una función interna que **recuerda** los valores de variables del entorno en el que fue creada, aunque ese entorno ya no exista.

19.3.1 Ejemplo básico de closure

```
def crear_saludo(nombre):
    def saludar():
        print(f"Hola, {nombre}")
    return saludar

saludo_luis = crear_saludo("Luis")
saludo_ana = crear_saludo("Ana")

saludo_luis()
saludo_ana()
```

Aunque `crear_saludo` ya terminó, las funciones `saludo_luis` y `saludo_ana` recuerdan el valor de `nombre`.

19.3.2 Closure con estado interno

```
def contador():
    cuenta = 0
```

```
    def incrementar():
        nonlocal cuenta
        cuenta += 1
        return cuenta

    return incrementar

c = contador()
print(c())  # 1
print(c())  # 2
print(c())  # 3
```

- `nonlocal cuenta` indica que `cuenta` no es local a `incrementar`, sino a la función externa `contador`.
- El closure mantiene el estado de `cuenta` entre llamadas.

19.3.3 Closure como fábrica de funciones

```
def potencia(exponente):
    def elevar(base):
        return base ** exponente
    return elevar

cuadrado = potencia(2)
cubo = potencia(3)

print(cuadrado(5))  # 25
print(cubo(2))      # 8
```

19.4 Funciones de orden superior

Una **función de orden superior** es una función que:

- Recibe una o más funciones como argumentos, o
- Devuelve una función.

Ya has visto muchos ejemplos en 19.1, 19.2 y 19.3. Python también incluye funciones de orden superior en su librería estándar.

19.4.1 map, filter, sorted con funciones

map(función, iterable) aplica una función a cada elemento:

```
def cuadrado(x):
    return x * x

numeros = [1, 2, 3, 4]
resultado = list(map(cuadrado, numeros))
print(resultado)  # [1, 4, 9, 16]
```

filter(función, iterable) filtra elementos según una condición:

```
def es_par(x):
    return x % 2 == 0

numeros = [1, 2, 3, 4, 5, 6]
pares = list(filter(es_par, numeros))
print(pares)  # [2, 4, 6]
```

sorted(iterable, key=función) ordena usando una función clave:

```
palabras = ["python", "es", "genial"]
ordenadas = sorted(palabras, key=len)
print(ordenadas)  # ['es', 'python', 'genial']
```

19.4.2 Funciones de orden superior propias

```
def aplicar_a_lista(funcion, lista):
    return [funcion(x) for x in lista]

def doble(x):
    return x * 2

datos = [1, 2, 3]
print(aplicar_a_lista(doble, datos))  # [2, 4, 6]
```

19.4.3 Combinando decoradores, closures y orden superior

```
from functools import wraps

def cache(funcion):
```

```
    cache_interna = {}

    @wraps(funcion)
    def envoltura(*args):
        if args in cache_interna:
            print("Devolviendo resultado cacheado")
            return cache_interna[args]
        print("Calculando resultado...")
        resultado = funcion(*args)
        cache_interna[args] = resultado
        return resultado

    return envoltura

@cache
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(10))
print(fib(10))  # esta vez sale de la caché
```

Aquí:

- `cache` es un decorador.
- Usa un **closure** (`cache_interna`) para recordar resultados.
- `fib` es una función recursiva decorada.

19.5 Buenas prácticas avanzadas

19.5.1 Funciones pequeñas y específicas

- Una función debe hacer **una sola cosa** y hacerla bien.
- Si una función crece demasiado, probablemente se pueda dividir.

19.5.2 Evitar efectos secundarios inesperados

- Una **función pura**: siempre devuelve lo mismo para los mismos argumentos y no modifica nada fuera de ella.
- Evita modificar listas o diccionarios que te pasan como parámetros, salvo que sea claramente la intención.

```
# Mejor: devolver una nueva lista
def duplicar_lista(lista):
    return [x * 2 for x in lista]
```

19.5.3 Documentar con docstrings

```
def media(lista):
    """
    Calcula la media aritmética de una lista de números.

    Parámetros:
        lista (list): lista de números (int o float).

    Devuelve:
        float: media de los valores. Si la lista está vacía,
devuelve 0.
    """
    if not lista:
        return 0
    return sum(lista) / len(lista)
```

19.5.4 Tipado opcional (type hints)

```
from typing import List

def media(lista: List[float]) -> float:
    if not lista:
        return 0.0
    return sum(lista) / len(lista)
```

No cambia el comportamiento, pero ayuda a herramientas y a otros programadores.

19.5.5 Decoradores bien diseñados

- Usar `functools.wraps` para no perder nombre y docstring.

- Asegurarse de que el decorador no rompe la firma de la función (usar `*args`, `**kwargs`).
- No abusar de decoradores si complican la lectura.

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Crea una función `saludar(nombre)` y asígnala a otra variable `f`. Llama a `f("Ana")`.

Ejercicio 2:
Crea una lista de funciones `[abs, float, str]` y aplícalas a un mismo valor, mostrando los resultados.

Ejercicio 3:
Crea un diccionario `operaciones` que asocie `"sumar"` y `"multiplicar"` a funciones que realicen esas operaciones. Pide al usuario una operación y dos números y ejecuta la función correspondiente.

Ejercicio 4:
Escribe una función `aplicar_operacion(a, b, operacion)` que reciba una función y la aplique a `a` y `b`. Prueba con `sumar` y `restar`.

Ejercicio 5:
Crea un decorador `traza` que imprima `"Llamando a FUNCION"` antes de ejecutar la función y `"Fin de FUNCION"` después. Aplícalo a una función `saludar()`.

Ejercicio 6:
Modifica el decorador `traza` para que muestre también los argumentos con los que se llama a la función.

Ejercicio 7:
Crea un decorador `tiempo` que mida el tiempo de ejecución de una función usando `time.time()`. Aplícalo a una función que calcule la suma de los números del 1 al 1_000_000.

Ejercicio 8:
Crea un decorador `mayusculas` que convierta a mayúsculas la cadena devuelta por una función que devuelve texto.

Ejercicio 9:
Crea una función `crear_saludo(nombre)` que devuelva una función que saluda a ese nombre (closure). Prueba con varios nombres.

Ejercicio 10:
Crea una función `contador()` que devuelva una función que, cada vez que se llame, incremente y devuelva un contador interno (closure con `nonlocal`).

Ejercicio 11:
Crea una función `potencia(exponente)` que devuelva una función que eleve un número a ese exponente. Prueba con `cuadrado` y `cubo`.

Ejercicio 12:
Usa `map` con una función `cuadrado` para obtener los cuadrados de `[1, 2, 3, 4, 5]`.

Ejercicio 13:
Usa `filter` con una función `es_par` para obtener los pares de `[1, 2, 3, 4, 5, 6, 7, 8]`.

Ejercicio 14:
Usa `sorted` con `key=len` para ordenar una lista de palabras por longitud.

Ejercicio 15:
Crea una función de orden superior `aplicar_a_lista(funcion, lista)` que devuelva una nueva lista con la función aplicada a cada elemento. Prueba con una función `doble`.

Ejercicio 16:
Crea un decorador `cache` sencillo que guarde en un diccionario los resultados de una función `f(n)` y devuelva el valor cacheado si se llama de nuevo con el mismo `n`.

Ejercicio 17:
Aplica el decorador `cache` a una función recursiva `fib(n)` y comprueba que la segunda llamada a `fib(30)` es mucho más rápida.

Ejercicio 18:
Escribe una función `media(lista)` bien documentada con docstring y type hints. Pruébala con varias listas.

Ejercicio 19:
Crea una función `es_pura(funcion, argumentos)` que llame varias veces a `funcion(*argumentos)` y compruebe si siempre devuelve lo mismo (muy simple). Pruébala con una función pura y otra que use una variable global.

Ejercicio 20:
Diseña un pequeño “mini-framework” de validación:

- Una función `validar_no_vacio(texto)` que devuelva `True/False`.
- Una función de orden superior `aplicar_validaciones(texto, validaciones)` que reciba una lista de funciones de validación y devuelva `True` solo si todas pasan.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
def saludar(nombre):
    print(f"Hola, {nombre}")


f = saludar
f("Ana")
```

Solución Ejercicio 2

```
funciones = [abs, float, str]
valor = -3

for f in funciones:
    print(f.__name__, "->", f(valor))
```

Solución Ejercicio 3

```
def sumar(a, b):
    return a + b

def multiplicar(a, b):
    return a * b

operaciones = {
    "sumar": sumar,
    "multiplicar": multiplicar
}

op = input("Operación (sumar/multiplicar): ")
a = float(input("a: "))
b = float(input("b: "))

if op in operaciones:
    resultado = operaciones[op](a, b)
    print("Resultado:", resultado)
else:
    print("Operación no válida")
```

Solución Ejercicio 4

```
def sumar(a, b):
    return a + b
```

```
def restar(a, b):
    return a - b

def aplicar_operacion(a, b, operacion):
    return operacion(a, b)

print(aplicar_operacion(10, 5, sumar))
print(aplicar_operacion(10, 5, restar))
```

Solución Ejercicio 5

```
def traza(funcion):
    def nueva_funcion():
        print(f"Llamando a {funcion.__name__}")
        funcion()
        print(f"Fin de {funcion.__name__}")
    return nueva_funcion

@traza
def saludar():
    print("Hola")

saludar()
```

Solución Ejercicio 6

```
def traza(funcion):
    def nueva_funcion(*args, **kwargs):
        print(f"Llamando a {funcion.__name__} con args={args},
kwargs={kwargs}")
        resultado = funcion(*args, **kwargs)
        print(f"Fin de {funcion.__name__}, resultado={resultado}")
        return resultado
```

```
    return nueva_funcion

@traza
def sumar(a, b):
    return a + b

s = sumar(3, 4)
print("Resultado final:", s)
```

Solución Ejercicio 7

```
import time

def tiempo(funcion):
    def nueva_funcion(*args, **kwargs):
        inicio = time.time()
        resultado = funcion(*args, **kwargs)
        fin = time.time()
        print(f"Tiempo de ejecución de {funcion.__name__}: {fin - inicio:.4f} segundos")
        return resultado
    return nueva_funcion

@tiempo
def suma_grande():
    total = 0
    for i in range(1, 1_000_001):
        total += i
    return total

print("Resultado:", suma_grande())
```

Solución Ejercicio 8

```
def mayusculas(funcion):
    def nueva_funcion(*args, **kwargs):
        texto = funcion(*args, **kwargs)
        return texto.upper()
    return nueva_funcion

@mayusculas
def obtener_mensaje():
    return "hola mundo"

print(obtener_mensaje())  # "HOLA MUNDO"
```

Solución Ejercicio 9

```
def crear_saludo(nombre):
    def saludar():
        print(f"Hola, {nombre}")
    return saludar

saludo_luis = crear_saludo("Luis")
saludo_ana = crear_saludo("Ana")

saludo_luis()
saludo_ana()
```

Solución Ejercicio 10

```
def contador():
    cuenta = 0

    def incrementar():
        nonlocal cuenta
        cuenta += 1
```

```
        return cuenta

    return incrementar

c = contador()
print(c())  # 1
print(c())  # 2
print(c())  # 3
```

Solución Ejercicio 11

```
def potencia(exponente):
    def elevar(base):
        return base ** exponente
    return elevar

cuadrado = potencia(2)
cubo = potencia(3)

print(cuadrado(5))  # 25
print(cubo(2))      # 8
```

Solución Ejercicio 12

```
def cuadrado(x):
    return x * x

numeros = [1, 2, 3, 4, 5]
resultado = list(map(cuadrado, numeros))
print(resultado)
```

Solución Ejercicio 13

```
def es_par(x):
    return x % 2 == 0
```

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8]
pares = list(filter(es_par, numeros))
print(pares)
```

Solución Ejercicio 14

```
palabras = ["python", "es", "genial", "super"]
ordenadas = sorted(palabras, key=len)
print(ordenadas)
```

Solución Ejercicio 15

```
def aplicar_a_lista(funcion, lista):
    return [funcion(x) for x in lista]

def doble(x):
    return x * 2

datos = [1, 2, 3]
print(aplicar_a_lista(doble, datos))
```

Solución Ejercicio 16

```
def cache(funcion):
    cache_interna = {}

    def nueva_funcion(n):
        if n in cache_interna:
            print("Devolviendo valor cacheado para", n)
            return cache_interna[n]
        print("Calculando valor para", n)
        resultado = funcion(n)
        cache_interna[n] = resultado
        return resultado
```

```
    return nueva_funcion

@cache
def cuadrado(n):
    return n * n

print(cuadrado(4))
print(cuadrado(4))  # esta vez sale de la caché
```

Solución Ejercicio 17

```
from functools import lru_cache

# O bien nuestro propio cache, pero aquí usamos lru_cache por comodidad
@lru_cache(maxsize=None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(30))
print(fib(30))  # segunda llamada muy rápida
```

(Con tu propio decorador `cache` también funcionaría, similar al ejercicio 16.)

Solución Ejercicio 18

```
from typing import List

def media(lista: List[float]) -> float:
    """
    Calcula la media aritmética de una lista de números.
```

```
    Parámetros:
        lista (list[float]): lista de números.

    Devuelve:
        float: media de los valores. Si la lista está vacía,
devuelve 0.0.
    """
    if not lista:
        return 0.0
    return sum(lista) / len(lista)

print(media([1, 2, 3, 4, 5]))
print(media([]))
```

Solución Ejercicio 19

```
contador_global = 0

def funcion_pura(x):
    return x * x

def funcion_impura(x):
    global contador_global
    contador_global += 1
    return x + contador_global

def es_pura(funcion, argumentos, repeticiones=3):
    resultados = [funcion(*argumentos) for _ in
range(repeticiones)]
    return len(set(resultados)) == 1

print("funcion_pura es pura?", es_pura(funcion_pura, (5,)))
print("funcion_impura es pura?", es_pura(funcion_impura, (5,)))
```

Solución Ejercicio 20

```
def validar_no_vacio(texto):
    return len(texto.strip()) > 0

def validar_longitud_minima(minimo):
    def validador(texto):
        return len(texto) >= minimo
    return validador

def aplicar_validaciones(texto, validaciones):
    for v in validaciones:
        if not v(texto):
            return False
    return True

texto = input("Introduce un texto: ")

validaciones = [
    validar_no_vacio,
    validar_longitud_minima(5)
]

if aplicar_validaciones(texto, validaciones):
    print("Texto válido")
else:
    print("Texto NO válido")
```

ANEXO

A0. Introducción

En muchos programas de consola queremos ir un poco más allá del simple `print()`:

- Poner un **título** a la ventana de la consola (especialmente en Windows).
- **Limpiar** la pantalla para que el usuario vea solo la información relevante.
- **Colocar texto** en posiciones concretas (coordenadas X/Y) para crear interfaces más ordenadas.
- Controlar la **aleatoriedad** mediante la **semilla** para poder reproducir resultados.

Este anexo recoge utilidades prácticas:

- **A1. Poner título a la ventana de la consola**
- **A2. Limpiar la consola**
- **A3. Colocar una cadena en una posición concreta**
- **A4. Randomize (semilla y aleatoriedad controlada)**

Con muchos ejemplos y, al final, **20 ejercicios con solución**.

A1. Poner título a la ventana de la consola

A1.1 Título en Windows

En **Windows**, la ventana de la consola (CMD o PowerShell) tiene un título que se puede cambiar. Desde Python, hay varias formas:

1) Usando os.system("title ...")

El comando de Windows para cambiar el título es `title`. Podemos llamarlo desde Python con `os.system`:

```
import os


os.system("title Mi Programa en Python")
print("Título cambiado.")
input("Pulsa ENTER para salir...")
```

- `os.system("title Mi Programa en Python")` ejecuta el comando `title` en la consola.
- El título de la ventana pasa a ser “Mi Programa en Python”.

2) Usando ctypes (más avanzado)

Hay formas más avanzadas usando la API de Windows, pero para la mayoría de programas de consola, `os.system("title ...")` es suficiente y sencillo.

3) Buenas prácticas con el título

- Usar títulos descriptivos: **“Gestor de Ficheros v1.0”**, **“Juego de Adivinanzas”**, etc.

- Cambiar el título al inicio del programa y, si quieres, actualizarlo según el estado (por ejemplo, “Procesando datos…”).

Ejemplo más completo:

```
import os
import time

os.system("title Gestor de Tareas - Python")

print("Iniciando gestor de tareas...")
time.sleep(2)

os.system("title Gestor de Tareas - Procesando...")
print("Procesando datos...")
time.sleep(2)

os.system("title Gestor de Tareas - Finalizado")
print("Proceso completado.")
```

A2. Limpiar la consola

A2.1 Limpieza en Windows

En Windows, el comando para limpiar la pantalla es `cls`. En sistemas tipo Unix (Linux, macOS), el comando es `clear`.

Podemos escribir una función que detecte el sistema operativo y use el comando adecuado.

Ejemplo básico para Windows:

```
import os

os.system("cls")  # Limpia la pantalla en Windows
print("Pantalla limpia.")
```

Ejemplo multiplataforma (Windows / Linux / macOS):

```
import os
import platform

def limpiar_pantalla():
```

```
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")

print("Texto inicial...")
input("Pulsa ENTER para limpiar la pantalla...")
limpiar_pantalla()
print("Pantalla limpia.")
```

- `platform.system()` devuelve `"Windows"`, `"Linux"`, `"Darwin"` (macOS), etc.
- Según el sistema, se ejecuta `cls` o `clear`.

Uso típico en programas interactivos:

- Mostrar un menú.
- Leer opción.
- Limpiar pantalla.
- Mostrar nueva información.

Ejemplo:

```
import os
import platform

def limpiar_pantalla():
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")

while True:
    limpiar_pantalla()
    print("=== MENÚ PRINCIPAL ===")
    print("1. Opción uno")
    print("2. Opción dos")
```

```
    print("3. Salir")
    opcion = input("Elige una opción: ")

    if opcion == "1":
        print("Has elegido la opción uno.")
    elif opcion == "2":
        print("Has elegido la opción dos.")
    elif opcion == "3":
        print("Saliendo...")
        break
    else:
        print("Opción no válida.")

    input("Pulsa ENTER para continuar...")
```

A3. Colocar una cadena en una posición concreta

Para colocar texto en una posición concreta de la consola, se usan **códigos ANSI de movimiento de cursor**. Esto funciona bien en muchas consolas (Linux, macOS, Windows moderno con soporte ANSI).

A3.1 Códigos ANSI de movimiento

El código general para mover el cursor a una posición (fila, columna) es:

`\033[<fila>;<columna>H`

- `\033` es el carácter ESC (escape).
- `[<fila>;<columna>H` indica la posición.
- Las filas y columnas empiezan en 1 (no en 0).

Ejemplo:

```
print("\033[5;10HTexto en fila 5, columna 10")
```

Esto mueve el cursor a la fila 5, columna 10, y luego imprime “Texto en fila 5, columna 10”.

A3.2 Coordenadas X/Y

En consola, normalmente:

- **Fila** (Y) → posición vertical (1 = arriba).
- **Columna** (X) → posición horizontal (1 = izquierda).

Podemos crear una función para escribir texto en una posición:

```
def escribir_en(x, y, texto):
    # x = columna, y = fila
    print(f"\033[{y};{x}H{texto}", end="")

# Ejemplo de uso:
print("\033[2J")  # Limpia la pantalla (código ANSI para borrar)
escribir_en(10, 5, "Hola en (10,5)")
escribir_en(20, 10, "Otro texto en (20,10)")
print()  # Para bajar el cursor a la siguiente línea
```

- `\033[2J` limpia la pantalla (borrado completo).
- `end=""` evita que `print` añada salto de línea automático.

A3.3 Ejemplos prácticos

Ejemplo 1: dibujar una "tabla" en posiciones fijas

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

escribir_en(5, 2, "Nombre")
escribir_en(20, 2, "Edad")

escribir_en(5, 3, "Ana")
escribir_en(20, 3, "30")

escribir_en(5, 4, "Luis")
escribir_en(20, 4, "25")

print("\033[6;1H")  # mover el cursor a la fila 6, columna 1
print("Fin de la tabla.")
```

Ejemplo 2: "panel de estado" en la parte superior

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")
```

```
print("\033[2J")  # limpiar pantalla

escribir_en(1, 1, "ESTADO: OK")
escribir_en(1, 2, "USO DE MEMORIA: 45%")
escribir_en(1, 3, "USUARIOS CONECTADOS: 3")

print("\033[5;1H")  # mover cursor a fila 5
print("Aquí iría el contenido principal...")
```

Ejemplo 3: “animación” simple moviendo texto

```
import time

def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

for x in range(1, 40):
    escribir_en(x, 5, "O")
    time.sleep(0.05)
    escribir_en(x, 5, " ")  # borrar la O
```

A4. Randomize: semilla y aleatoriedad controlada

En muchos programas usamos números aleatorios (juegos, simulaciones, pruebas). Python tiene el módulo `random` para esto. Pero a veces queremos que la “aleatoriedad” sea **reproducible**: que, si ejecutamos el programa otra vez, obtengamos los mismos “aleatorios”.

Ahí entra la **semilla**.

A4.1 Inicialización de la semilla

La **semilla** es el valor inicial que usa el generador de números aleatorios. Si usamos la misma semilla, obtendremos la misma secuencia de números.

En Python, la semilla se establece con `random.seed(valor)`.

```
import random
```

```
random.seed(123)  # fijamos la semilla


print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
```

Si ejecutas este código varias veces, verás que los tres números son siempre los mismos (para esa semilla).

A4.2 Uso de random.seed()

Ejemplo: reproducir una simulación

```
import random


random.seed(42)


for i in range(5):
    print(random.random())
```

- `random.random()` devuelve un número entre 0 y 1.
- Con semilla fija, la secuencia es siempre igual.

Cambiar la semilla para obtener otra secuencia

```
import random


random.seed(1)
print("Secuencia con semilla 1:")
for i in range(3):
    print(random.randint(1, 100))


random.seed(2)
print("Secuencia con semilla 2:")
for i in range(3):
    print(random.randint(1, 100))
```

A4.3 Generación controlada de valores aleatorios

Ejemplo: juego reproducible

```
import random
```

```
def jugar(semilla):
    random.seed(semilla)
    numero_secreto = random.randint(1, 10)
    print("He pensado un número entre 1 y 10.")
    intento = int(input("Adivina: "))
    if intento == numero_secreto:
        print("¡Has acertado!")
    else:
        print("No, el número era", numero_secreto)

# Si llamas varias veces con la misma semilla, el número secreto será el mismo
jugar(123)
```

Ejemplo: pruebas automáticas

En pruebas (tests), es útil fijar la semilla para que los resultados sean siempre iguales:

```
import random

def generar_lista_aleatoria(n):
    return [random.randint(1, 100) for _ in range(n)]

random.seed(0)
lista1 = generar_lista_aleatoria(5)

random.seed(0)
lista2 = generar_lista_aleatoria(5)

print(lista1)
print(lista2)
print("¿Son iguales?", lista1 == lista2)
```

---

20 EJERCICIOS PROPUESTOS (SOLUCIONES AL FINAL)

Ejercicio 1:
Escribe un programa que ponga el título de la ventana de la consola en Windows a `"Mi Primer Programa"` usando `os.system("title ...")`.

Ejercicio 2:
Modifica el programa anterior para que, después de poner el título, muestre un mensaje y espere a que el usuario pulse ENTER antes de terminar.

Ejercicio 3:
Crea una función `poner_titulo(titulo)` que reciba una cadena y ponga ese título en la ventana de la consola (Windows).

Ejercicio 4:
Escribe una función `limpiar_pantalla()` que use `platform.system()` para decidir si debe ejecutar `cls` (Windows) o `clear` (otros sistemas).

Ejercicio 5:
Crea un pequeño menú que, antes de mostrar cada pantalla, llame a `limpiar_pantalla()` para que la consola esté limpia.

Ejercicio 6:
Escribe una función `escribir_en(x, y, texto)` que coloque `texto` en la posición `(x, y)` usando códigos ANSI. Pruébala colocando tres textos en distintas posiciones.

Ejercicio 7:
Usando `escribir_en`, dibuja una "tabla" con cabecera y dos filas de datos en posiciones fijas.

Ejercicio 8:
Crea una "animación" simple donde una letra `*` se mueva de izquierda a derecha en la fila 5, borrándose en cada paso.

Ejercicio 9:
Escribe un programa que limpie la pantalla con `\033[2J` y luego coloque un mensaje en el centro aproximado de la consola (por ejemplo, fila 10, columna 30).

Ejercicio 10:
Crea una función `panel_estado()` que limpie la pantalla y coloque tres líneas de estado en las filas 1, 2 y 3, usando `escribir_en`.

Ejercicio 11:
Escribe un programa que use `random.seed(123)` y muestre tres números aleatorios entre 1 y 10. Ejecuta el programa varias veces y observa que los números se repiten.

Ejercicio 12:
Modifica el programa anterior para que pida al usuario una semilla (entero) y la use con `random.seed(semilla)` antes de generar los números.

Ejercicio 13:
Crea una función `generar_lista_aleatoria(n, semilla)` que fije la semilla y devuelva

una lista de `n` números aleatorios entre 1 y 100. Pruébala con la misma semilla y comprueba que las listas son iguales.

Ejercicio 14:
Escribe un programa que simule lanzar un dado 10 veces usando `random.randint(1, 6)` con una semilla fija. Muestra los resultados.

Ejercicio 15:
Crea una función `jugar(semilla)` que fije la semilla, piense un número entre 1 y 10 y pida al usuario que lo adivine. Prueba a llamar varias veces con la misma semilla.

Ejercicio 16:
Combina `limpiar_pantalla()` y `escribir_en` para crear una pantalla de "bienvenida" con un título en la parte superior y un mensaje centrado.

Ejercicio 17:
Crea una función `dibujar_cuadro(x1, y1, x2, y2)` que dibuje un rectángulo usando caracteres # en las posiciones correspondientes (usando `escribir_en`).

Ejercicio 18:
Escribe un programa que muestre un "mapa" simple (por ejemplo, un área de 20x10) y coloque un símbolo @ en una posición aleatoria usando `random.randint` y `escribir_en`.

Ejercicio 19:
Crea una función `semilla_aleatoria()` que fije la semilla usando el tiempo actual (`time.time()`) convertido a entero, y luego genere un número aleatorio entre 1 y 100.

Ejercicio 20:
Diseña un pequeño programa que combine todo:

- Poner título a la ventana.
- Limpiar pantalla.
- Mostrar un panel de estado en la parte superior.
- Colocar un símbolo en una posición aleatoria.
- Usar una semilla fija para que el símbolo aparezca siempre en la misma posición si se repite la ejecución.

SOLUCIONES A LOS 20 EJERCICIOS

Solución Ejercicio 1

```
import os


os.system("title Mi Primer Programa")
print("Título cambiado a 'Mi Primer Programa'.")
```

Solución Ejercicio 2

```
import os

os.system("title Mi Primer Programa")
print("Título cambiado a 'Mi Primer Programa'.")
input("Pulsa ENTER para terminar...")
```

Solución Ejercicio 3

```
import os

def poner_titulo(titulo):
    os.system(f"title {titulo}")

poner_titulo("Gestor de Ficheros")
print("Título cambiado.")
```

Solución Ejercicio 4

```
import os
import platform

def limpiar_pantalla():
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")

print("Texto inicial...")
input("Pulsa ENTER para limpiar...")
limpiar_pantalla()
print("Pantalla limpia.")
```

Solución Ejercicio 5

```
import os
import platform

def limpiar_pantalla():
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")

while True:
    limpiar_pantalla()
    print("=== MENÚ ===")
    print("1. Opción uno")
    print("2. Opción dos")
    print("3. Salir")
    opcion = input("Elige opción: ")

    if opcion == "1":
        print("Has elegido la opción uno.")
    elif opcion == "2":
        print("Has elegido la opción dos.")
    elif opcion == "3":
        print("Saliendo...")
        break
    else:
        print("Opción no válida.")

    input("Pulsa ENTER para continuar...")
```

Solución Ejercicio 6

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

escribir_en(5, 3, "Texto en (5,3)")
escribir_en(10, 5, "Texto en (10,5)")
escribir_en(20, 7, "Texto en (20,7)")

print("\033[9;1H")  # mover cursor a fila 9
print("Fin del ejemplo.")
```

Solución Ejercicio 7

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

# Cabecera
escribir_en(5, 2, "Nombre")
escribir_en(20, 2, "Edad")

# Fila 1
escribir_en(5, 3, "Ana")
escribir_en(20, 3, "30")

# Fila 2
escribir_en(5, 4, "Luis")
escribir_en(20, 4, "25")

print("\033[6;1H")
```

```
print("Tabla completa.")
```

Solución Ejercicio 8

```
import time

def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

for x in range(1, 40):
    escribir_en(x, 5, "*")
    time.sleep(0.05)
    escribir_en(x, 5, " ")
```

Solución Ejercicio 9

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

# Centro aproximado: fila 10, columna 30
escribir_en(30, 10, "MENSAJE CENTRADO")
print("\033[12;1H")
print("Fin.")
```

Solución Ejercicio 10

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

def panel_estado():
    print("\033[2J")  # limpiar pantalla
```

```
    escribir_en(1, 1, "ESTADO: OK")
    escribir_en(1, 2, "USO CPU: 23%")
    escribir_en(1, 3, "USUARIOS: 5")
    print("\033[5;1H")

panel_estado()
print("Contenido principal debajo del panel...")
```

Solución Ejercicio 11

```
import random

random.seed(123)

print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
```

Solución Ejercicio 12

```
import random

semilla = int(input("Introduce una semilla (entero): "))
random.seed(semilla)

print("Tres números aleatorios entre 1 y 10:")
print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
```

Solución Ejercicio 13

```
import random

def generar_lista_aleatoria(n, semilla):
```

```
    random.seed(semilla)
    return [random.randint(1, 100) for _ in range(n)]

lista1 = generar_lista_aleatoria(5, 42)
lista2 = generar_lista_aleatoria(5, 42)

print(lista1)
print(lista2)
print("¿Son iguales?", lista1 == lista2)
```

Solución Ejercicio 14

```
import random

random.seed(10)

print("Lanzamientos de dado:")
for i in range(10):
    print(random.randint(1, 6), end=" ")

print()
```

Solución Ejercicio 15

```
import random

def jugar(semilla):
    random.seed(semilla)
    numero_secreto = random.randint(1, 10)
    print("He pensado un número entre 1 y 10.")
    intento = int(input("Adivina: "))
    if intento == numero_secreto:
        print("¡Has acertado!")
    else:
```

```
        print("No, el número era", numero_secreto)

jugar(123)
```

Solución Ejercicio 16

```
import os
import platform

def limpiar_pantalla():
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")

def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

limpiar_pantalla()
print("\033[2J")  # limpiar con ANSI también

escribir_en(1, 1, "=== BIENVENIDA ===")
escribir_en(20, 10, "Bienvenido a mi programa en consola")
print("\033[12;1H")
print("Pulsa ENTER para continuar...")
input()
```

Solución Ejercicio 17

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

def dibujar_cuadro(x1, y1, x2, y2):
```

```
    # Líneas horizontales
    for x in range(x1, x2 + 1):
        escribir_en(x, y1, "#")
        escribir_en(x, y2, "#")
    # Líneas verticales
    for y in range(y1, y2 + 1):
        escribir_en(x1, y, "#")
        escribir_en(x2, y, "#")

print("\033[2J")  # limpiar pantalla
dibujar_cuadro(5, 3, 30, 10)
print("\033[12;1H")
print("Cuadro dibujado.")
```

Solución Ejercicio 18

```
import random

def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

print("\033[2J")  # limpiar pantalla

# Área 20x10, filas 2-11, columnas 2-21
for y in range(2, 12):
    for x in range(2, 22):
        escribir_en(x, y, ".")

# Posición aleatoria
random.seed(50)
x_random = random.randint(2, 21)
y_random = random.randint(2, 11)
```

```
escribir_en(x_random, y_random, "@")

print("\033[13;1H")
print("Mapa con símbolo @ en posición aleatoria.")
```

Solución Ejercicio 19

```
import random
import time

def semilla_aleatoria():
    semilla = int(time.time())
    random.seed(semilla)
    return random.randint(1, 100)

print("Número aleatorio con semilla basada en tiempo:",
semilla_aleatoria())
```

Solución Ejercicio 20

```
import os
import platform
import random

def poner_titulo(titulo):
    os.system(f"title {titulo}")

def limpiar_pantalla():
    sistema = platform.system()
    if sistema == "Windows":
        os.system("cls")
    else:
        os.system("clear")
```

```
def escribir_en(x, y, texto):
    print(f"\033[{y};{x}H{texto}", end="")

def panel_estado():
    escribir_en(1, 1, "ESTADO: OK")
    escribir_en(1, 2, "MODO: DEMO")
    escribir_en(1, 3, "SEMILLA: 123")

def main():
    poner_titulo("Demo Consola y Aleatoriedad")
    limpiar_pantalla()
    print("\033[2J")  # limpiar con ANSI

    panel_estado()

    # Semilla fija
    random.seed(123)
    x = random.randint(10, 40)
    y = random.randint(5, 20)

    escribir_en(x, y, "@")

    print("\033[22;1H")
    print("Símbolo @ colocado en posición aleatoria
reproducible.")
    input("Pulsa ENTER para salir...")

main()
```